НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«На правах рукопису»
УДК 004.9

«До захисту допущено»
В.о. завідувача кафедри
_____ О.І. Ролік
«___»_____2018 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 151 Автоматизація та комп'ютерно-інтегровані технології

на тему: «Бібліотеки машинного навчання для розв'язку задач розпізнавання»

Виконала:
студентка ІІ курсу, групи ІА-61м
Левченко Ксенія В'ячеславівна _____

Керівник:
Доцент, к.т.н, доцент
Дорогий Я.Ю. _____

Рецензент:
Доц. каф. кібербезпеки та ЗІСТ, к.т.н., доцент
Цуркан В.В. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2018 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – другий (магістерський) за освітньо-науковою програмою

Спеціальність – 151 «Автоматизація та комп'ютерно-інтегровані технології»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ С.Ф. Теленик

«___»_____20__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Левченко Ксенії В'ячеславівні

1. Тема дисертації «Бібліотеки машинного навчання для розв'язку задач розпізнавання», науковий керівник дисертації Дорогий Ярослав Юрійович, доцент, к.т.н., доцент, затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом дисертації _____

3. Об'єкт дослідження – бібліотеки машинного навчання.

4. Предмет дослідження – компаративний аналіз бібліотек машинного навчання за визначеними критеріями.

5. Перелік завдань, які потрібно розробити: вибір бібліотек машинного навчання; аналіз існуючих критеріїв порівняння; визначення необхідного набору критеріїв порівняння; проведення порівняння за обраними критеріями; аналіз залежності між новими та старими критеріями; написання пояснювальної записки.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: структура побудованої нейронної мережі; графік часу навчання нейронної мережі; графік часу роботи нейронної мережі за новим критерієм.

7. Перелік публікацій: 1) «Порівняння фреймворків машинного навчання» – VI Міжнародна науково-практична конференція «Summer InfoCom Advanced Solutions 2018» 2) «Порівняння часу виконання базових операцій фреймворків машинного

навчання» – Науково-практичний журнал «Інфокомунікаційні системи та технології» – 2018 – №2(2). – С. 27 – 31.

8. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вибір бібліотек машинного навчання	15.03.18-29.03.18	
2	Аналіз існуючих критеріїв порівняння	30.03.18-09.04.18	
3	Визначення необхідного набору критеріїв порівняння	10.04.18-17.04.18	
4	Проведення порівняння за обраними критеріями	18.04.18-28.04.18	
5	Аналіз залежності між новими та старими критеріями	29.04.18-02.05.18	
6	Написання пояснювальної записки	03.05.18-14.05.18	

Студент К.В. Левченко

Науковий керівник дисертації Я.Ю. Дорогий

РЕФЕРАТ

Магістерська дисертація освітньо-кваліфікаційного рівня “магістр” на тему “Бібліотеки машинного навчання для розв’язку задач розпізнавання”: 118с., 63 рис., 28 табл., 2 додатки, 24 джерела.

Об'єкт дослідження – бібліотеки машинного навчання.

Мета роботи – оцінити бібліотеки машинного навчання за вже існуючими критеріями та за новими, які представлені у цій роботі.

Бібліотеки машинного навчання використовуються у всіх системах, що використовують нейронні мережі, тому важливо правильно підібрати бібліотеку для розв’язку своєї задачі. Саме тому у даній роботі найпопулярніші бібліотеки TensorFlow, PyTorch, MXNet, CNTK, Caffe були порівняні за часом тренування і виконання базових операцій, за способом написання моделі нейронної мережі, підтримкою попередньо натренованих моделей, підтримкою графічного процесора тощо.

Для проведення експериментів використовувалась мова програмування Python 3.6.4.

Прогнозні припущення про розвиток дослідження – побудова інших видів нейронних мереж для оцінки роботи бібліотек з ними.

МАШИННЕ НАВЧАННЯ, БІБЛІОТЕКИ МАШИННОГО НАВЧАННЯ, НЕЙРОННІ МЕРЕЖІ, ЗАДАЧА РОЗПІЗНАВАННЯ, ІМПЕРАТИВНИЙ ТА СИМВОЛІЧНИЙ ПІДХОДИ ПРОГРАМУВАННЯ.

ABSTRACT

Master's thesis “Machine learning frameworks for pattern recognition” consists of 118 pages, 63 figures, 28 tables, 2 appendices, 24 sources.

The object of the study are machine learning frameworks.

The purpose of the work is to analyze machine learning frameworks by existing and new criteria.

Machine learning frameworks are used in every system, which use neural networks, so it’s important to choose a correct machine learning framework. That’s why the most popular frameworks TensorFlow, PyTorch, MXNet, CNTK, Caffe were used in this work. They were compared by training time, by model creation, by time of the execution of basic operations, by support of fine-tuning, by GPU support etc.

Python 3.6.4 was used to perform experiments.

Foreseeable assumptions about the development of the study – implementing new types of neural networks for rating work of the machine learning frameworks.

MACHINE LEARNING, MACHINE LEARNING FRAMEWORKS, NEURAL NETWORKS, PATTERN RECOGNITION, IMPERATIVE AND SYMBOLIC PROGRAMMING.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ОДИНИЦЬ І ТЕРМІНІВ

ШНМ – штучна нейронна мережа

CNN – Convolutional neural network (згорткова нейронна мережа)

ReLU – Rectified Linear Unit

SGD - Stochastic gradient descent (метод стохастичного градієнта)

ВСТУП

Тема, що розкривається в магістерській дисертації є актуальною, оскільки нейронні мережі використовуються все ширше, наприклад, у таких галузях як медична діагностика (виявлення вад на рентгенівських знімках), розпізнавання текстів, мови, технічна діагностика, системи контролю. Нейронні мережі дозволяють розв’язувати складні задачі класифікації, у яких без автоматизації було б задіяно велику кількість людських ресурсів. Очевидно, що для розробки таких систем краще використовувати вже готові рішення, які значно понижують поріг входження у галузь розробки систем розпізнавання та спрощують розробку в цілому. Бібліотеки машинного навчання полегшують реалізацію систем на основі нейронних мереж та дозволяють сфокусуватись на архітектурі системи, на задачах, що вона розв’язує, а не на розробці конкретного програмного забезпечення.

Існує велика кількість порівнянь бібліотек машинного навчання за різними критеріями: підтримка певних функцій, час роботи, зручність використання, підтримка тощо. Проте результати цих порівнянь не є структурованими та повними, в основному проведені за різних умов, що не дає можливості об’єктивно оцінити роботу кожного фреймворка.

Об’єктом дослідження є найпопулярніші бібліотеки машинного навчання, які порівнюються у цій роботі. Предметом дослідження є компаративний аналіз бібліотек машинного навчання за визначеними критеріями. Методами дослідження, що використовуються для досягнення мети є методи та моделі цифрової обробки зображень для підготовки вихідної бази зображень, методи розпізнавання образів для виконання ідентифікації зображень, методи обробки інформації й алгоритми навчання для створення та навчання ШНМ, алгоритми порівняння за обраними критеріями.

Зв’язок роботи з науковими програмами, планами, темами. Тематика роботи включена в науково-технічні плани кафедри автоматики та управління в технічних системах Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

Метою дослідження є визначення нових критеріїв, за якими можна порівняти бібліотеки машинного навчання. Важливо розуміти час роботи кожного фреймворка, коли він виконує базові математичні операції, оскільки бібліотеки машинного навчання використовуються не тільки для тренування нейронних мереж, а й розподілених обчислень над великою кількістю даних. Аналогічних порівнянь за цим критерієм не було знайдено. Для досягнення мети потрібно розв’язати наступні задачі: обрати бібліотеки для дослідження, провести компаративний аналіз за існуючими критеріями та за часом роботи базових операцій. Використані ті базові операції, які найчастіше виконуються під час тренування нейронної мережі.

Правильний вибір фреймворка для реалізації є важливим, оскільки швидкість його роботи прямо впливає на швидкість розробки. Зазвичай порівняння проводиться за критерієм часу, за який мережа досягне певної точності, проте це є довгим способом. Практична цінність даної роботи полягає в тому, що її результат допоможе швидше обирати бібліотеку для розв’язку конкретної задачі.

Новизна магістерського дослідження полягає у визначенні необхідного набору критеріїв оцінювання бібліотек машинного навчання, за допомогою якого оцінювати їх за часом роботи буде швидше, ніж при повноцінному тренуванні.

Результати дослідження апробовані і опубліковані у вигляді тез «Порівняння фреймворків машинного навчання» на VI Міжнародній науково-практичній конференції «Summer InfoCom Advanced Solutions 2018» (додаток А) та у вигляді статті «Порівняння часу виконання базових операцій фреймворків машинного навчання» у журналі «Інфокомунікаційні системи та технології» № 2(2) (додаток Б).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Машинне навчання

Машинне навчання - це галузь комп'ютерних наук, яка використовує статистичні методи, щоб дати комп'ютерним системам можливість "вчитися" (наприклад, поступово покращувати продуктивність у конкретному завданні) з даними, без явного використання програмування алгоритмів.

Машинне навчання використовується здебільше для вирішення проблем, які є занадто складними та потребують адаптації. Тобто це такий клас задач, який неможливо вирішити певним чітким алгоритмом, необхідно зважати на вже отримані результати. Прикладами таких задач є:

Завдання, що виконуються тваринами / людьми: є багато завдань, які люди, виконуємо регулярно, але наш аналіз відносно як ми робимо їх, недостатньо продуманий, щоб видобути чітко визначений алгоритм. Прикладами таких завдань є водіння, розпізнавання мови та розпізнавання образів. У всіх цих завданнях програми, що використовують машинне навчання, програм, які "навчаються з їхнього досвіду", досягають цілком задовільних результатів, коли навчаються на великій кількості тренувальних даних.

Завдання, що виходять за рамки людських можливостей: Ще один широкий клас завдань, що мають користь від машинного навчання, пов'язані з аналізом дуже великих і складних наборів даних: астрономічні дані, трансформація медичних архівів у медичні знання, прогноз погоди, двигун для веб-пошуку та електронної комерції. Із зростанням накопленої цифрової інформації стає очевидно, що серед всієї накопленої інформації є дійсно важлива, яку людина сама не може знайти, адже даних занадто багато і вони дуже складні. Навчитися виявляти значущі моделі в великих і складних наборах даних є перспективною областю, в якій поєднання програм, які навчаються з майже необмеженою ємністю пам’яті і постійно зростаючою швидкістю обробки даних, значно полегшує роботу з цим типом задач.

Адаптивність. Однією з обмежених можливостей звичайних програм є їх жорсткість – програма була написана і встановлена, далі вона залишається незмінною. Однак багато завдань змінюються з плином часу або від одного користувача до іншого. Інструменти машинного навчання - програми, поведінка яких пристосовується до їхніх вхідних даних - пропонують вирішення таких питань; вони, за своїм характером, є адаптивними до змін у середовищі, з яким вони взаємодіють. Типове успішне застосування машинного навчання включає програми, які розпізнають рукописний текст, де фіксована програма може адаптуватися до варіацій між почерком різних користувачів; програми виявлення спаму, автоматично адаптується до змін характеру спам-листів; програми розпізнавання мови.

Загалом, машинне навчання використовуються для розв’язку задачі класифікації: коли певний образ, вхідні дані необхідно віднести до того чи іншого класу, якщо вони вже відомі (навчання з учителем), і для розв’язку задачі кластеризації, де необхідно виділити певні ознаки, за якими можна розділити вхідні дані і згрупувати їх за цими ознаками (навчання без учителя). У даній роботі буде розглядатись задача класифікації.

Класифікатор (classifier) - це система, яка вводить (як правило) вектор дискретних та / або неперервних функцій і виводить одне дискретне значення класу. Наприклад, фільтр спаму класифікує повідомлення електронної пошти на "спам" або "не спам", і його вхідними даними може бути вектором булевих значень $x = (x_1, ..., x_j, ..., x_d)$, де $xj = 1$, якщо j-е слово в словнику з'являється в електронному листі, а $x_j = 0$ в іншому випадку. *Учень (learner)* вводить *навчальний набір (training set)* прикладів (x_i, y_i), де $x_i = (x_{i,1}, ..., x_{i,d})$ - це спостережуваний ввід, y_i - відповідний вихід і виводить класифікатор. Тест учня полягає в тому, чи дає цей класифікатор правильний вихід y_t для майбутніх прикладів x_t (наприклад, чи фільтр спаму правильно класифікує раніше невидимі електронні листи як спам чи не спам).

Машинне навчання проводиться за використанням різних алгоритмів, проте для всіх алгоритмів найважливішими є 3 компоненти:

Представлення. Класифікатор повинен бути представлений за допомогою формальної мови, яку комп'ютер може обробляти. І навпаки, вибір представлення для

учня рівносильний вибору набору класифікаторів, яким він може навчитися. Цей набір називається гіпотезою простору *(hypothesis space)* учня. Якщо класифікатор не знаходиться в гіпотезі простору, то він не можу бути вивчений.

Оцінка. Функція оцінки (також звана *цільова функція (objective function)* або *функція оцінки (scoring function)*) потрібна для виділення хороших класифікаторів від поганих. Функція оцінювання, яка використовується всередині алгоритму, може відрізнятись від зовнішньої, яку ми хочемо оптимізувати для класифікатора, для простоти оптимізації.

Оптимізація. Нарешті, нам потрібен метод пошуку серед класифікаторів того, який буде класифікувати найбільш швидко і правильно. Вибір методу оптимізації є ключовим елементом ефективності учня, а також допомагає визначити вибраний класифікатор, якщо функція оцінки має більше одного оптимуму. Для нових учнів найкраще почати використовувати загальноприйняті оптимізатори, які пізніше замінюються спеціально розробленими. [1]

1.2 Задача розпізнавання

Людина звичайно може легко розрізняти звук образи, звуки, аромати. Проте для комп'ютера важко вирішити такі проблеми сприйняття. Ці проблеми важкі, тому що кожен шаблон зазвичай містить велику кількість інформації, і проблеми розпізнавання зазвичай мають неясну, високовимірну структуру.

Розпізнавання образів - це наука робити висновки з даних, що були отримані шляхом сприйняття, використовуючи інструменти зі статистики, ймовірності, обчислювальної геометрії, машинного навчання, обробки сигналів та алгоритмів. Таким чином, це має найважливіше значення для штучного інтелекту та комп'ютерного бачення, і має далекосяжні застосування в галузі машинобудування, науки, медицини та бізнесу. Зокрема, досягнення, досягнуті протягом останнього півстоліття, дозволяють комп'ютерам ефективніше взаємодіяти з людьми та природним світом (наприклад, програмне забезпечення для розпізнавання мовлення). Проте найважливіші проблеми розпізнавання образів ще не вирішені. [2]

Цілком природно, що ми повинні прагнути спроектувати і побудувати машини, здатні розпізнавати шаблони. Від автоматичного розпізнавання мови, ідентифікації відбитків пальців, оптичного розпізнавання символів, ідентифікації послідовності ДНК та багато іншого, зрозуміло, що надійне, точне розпізнавання образів за допомогою машини буде надзвичайно корисним. Більше того, при вирішенні невизначеного числа проблем, необхідних для побудови таких систем, ми отримуємо глибше розуміння та оцінку систем розпізнавання образів. Для деяких завдань, таких як мова та візуальне розпізнавання, на наші проектні зусилля насправді можуть впливати знання про те, як вони вирішуються в природі, як в алгоритмах, які ми використовуємо, так і в дизайні спеціального устаткування. [3]

1.2.1 Основні поняття задачі розпізнавання

Ознаки (features) можна визначити як будь-який відмітний аспект, якість або характеристику, які можуть бути символічними (наприклад, колір) чи чисельними (наприклад, висота). Комбінація функцій d представляється як вектор d-розмірного стовпця, який називається вектором властивостей. D-мірний простір, визначений функцією вектора, називається *простором ознак (feature space)*. Об'єкти представлені у вигляді точок у просторі ознак. Це подання називається графіком розсіювання (scatter plot) [4].

Шаблон (pattern) визначається як комбінація ознак, характерних для особи. У класифікації шаблон - це пара змінних *{x, w}*, де *x* - це сукупність спостережень або ознак (вектор ознак), а *w* - концепція, що лежить в основі спостереження (label). Якість вектора властивостей пов'язана з його здатністю відрізняти приклади з різних класів (рис. 1.1). Приклади з того ж класу повинні мати подібні значення ознак, а приклади з різних класів мають різні значення ознак.

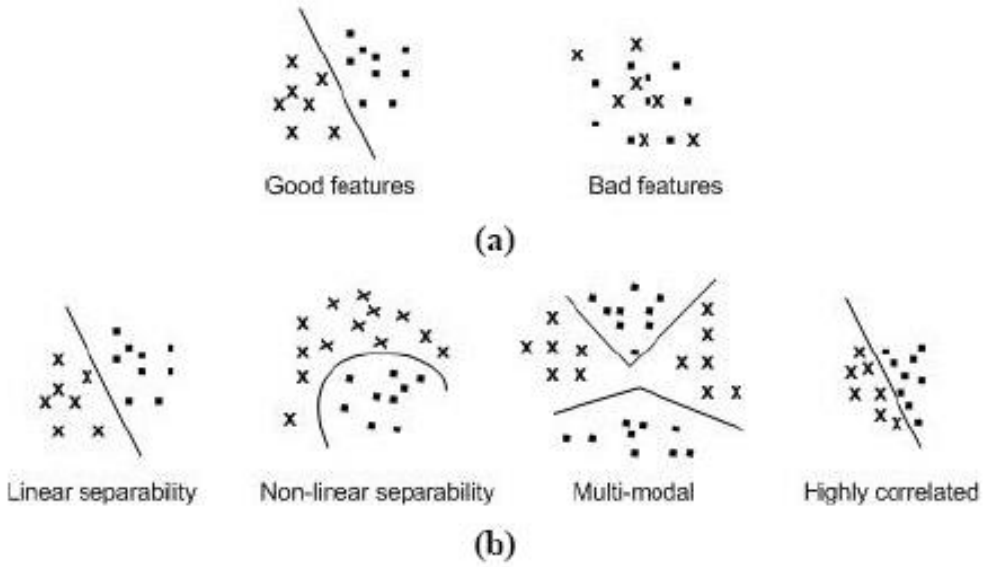


Рисунок 1.1 - Ознаки (характеристики): a - відмінність між хорошими (good) та поганими (bad) ознаками; b - властивості ознак за способом розділення: лінійні, нелінійні, мультимодальні, висококорельовані. [5]

Метою класифікатора є розділення простору ознак на позначені класом області прийняття рішень. Межі між регіонами прийняття рішень називаються межами рішень (рис. 1.2).

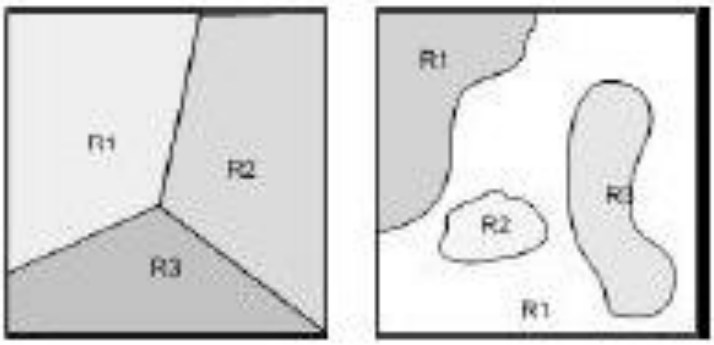


Рисунок 1.2 - Межі класифікатора та рішення. [5]

Якщо характеристики або атрибути класу відомі, окремі об'єкти можуть бути ідентифіковані як такі, що належать або не належать до цього класу. Об'єкти відносяться до класів, дотримуючись шаблонів відмінних характеристик і порівнюючи їх з типовими членами кожного класу. Розпізнавання образів передбачає

виділення образів з даних, їх аналізу та, нарешті, ідентифікації категорії (класу), до якої належить кожний образ. Типова система розпізнавання образів містить датчик, механізм попередньої обробки (сегментація), механізм виділення ознак (ручний або автоматичний), алгоритм класифікації або опису та набір прикладів (тренувальний набір), які вже класифіковані або описані (після обробки) (рис. 1.3).

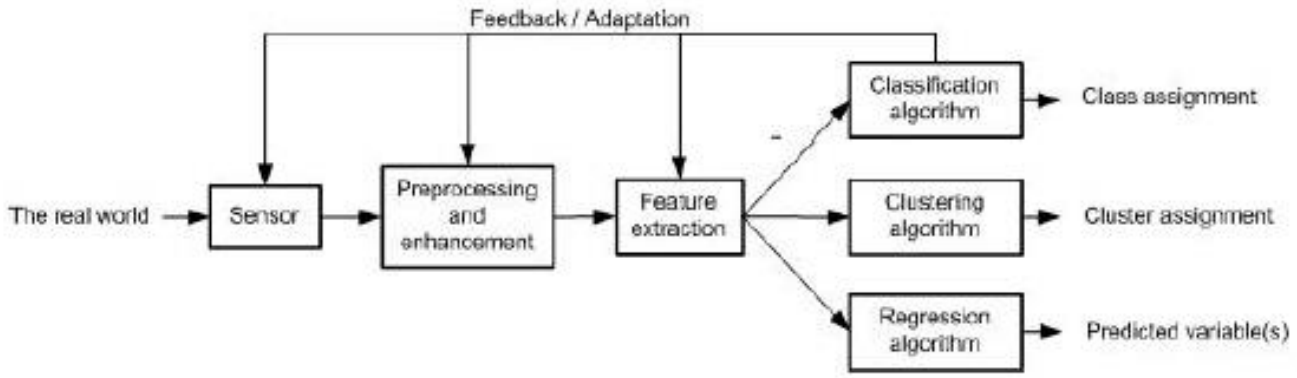


Рисунок 1.3 – Система розпізнавання образів [5]

1.2.2 Підходи до розпізнавання образів

Існує два фундаментальні підходи до впровадження системи розпізнавання образів: статистичний та структурний. Кожен підхід використовує різні методи для виконання описів та класифікації завдань. Гібридні підходи, іноді називаються уніфікованим підходом до розпізнавання образів, об'єднують як статистичні, так і структурні методики в рамках системи розпізнавання образів [6].

Визнання статистичного моделювання спирається на встановлені поняття в теорії статистичних рішень для того, щоб виокремити дані різних груп на основі кількісних характеристик даних. Існує широкий спектр статистичних методів, які можуть використовуватися в рамках завдання з виділення ознак, починаючи від простої описової статистики до складних перетворень. Приклади статистичних методів виділення ознак включають в себе середнє та стандартне відхилення обчислень, підсумовування частотних значень, перетворення Кархунена-Лоуєва, перетворення Фур'є, вейвлет-перетворення та трансформації Хога. Кількісні

характеристики, виділені з кожного об'єкта для статистичного розпізнавання образів, організовані у вектор-функцію з фіксованою довжиною, коли значення, пов'язане з кожною функцією, визначається його положенням у векторі (тобто перша ознака описує певну характеристику даних, друга ознака описує іншу характеристику тощо). Колекція векторів функцій, породжених завданням опису, передається до завдання класифікації. Статистичні методи, що використовуються в якості класифікаторів у межах задачі класифікації включають ті, які базуються на подібності (наприклад, відповідність шаблону, k-найближчого сусіда), ймовірність (наприклад, правило Байєса), межі (наприклад, дерева рішень, нейронні мережі) та кластеризація (наприклад, k-середні, ієрархічний).

Кількісний характер статистичного розпізнавання образів ускладнює виділення різниці між групами на основі морфологічних (тобто форми або структурних) підшаблонів та їх взаємозв'язків, вбудованих в дані. Це обмеження дало імпульс розвитку структурного підходу до розпізнавання образів, який підтверджується психологічними свідченнями, що стосуються функціонування людського сприйняття та пізнання. Розпізнавання об'єктів в людях було продемонстровано для залучення психічних уявлень до явних, структурно-орієнтованих характеристик об'єктів, і було зроблено висновок, що рішення про класифікацію людини складаються з урахуванням ступеня подібності між витягнутими ознаками та прототипом, розробленим для кожної групи. Наприклад, визнання за теорією компонентів пояснює процес розпізнавання образів у людей: (1) об'єкт сегментований у окремі області за границями, визначеними різними поверхневими характеристиками (наприклад, яскравістю, текстурою та кольором); (2) кожна сегментована область апроксимується простою геометричною формою, і (3) об'єкт визначається на підставі подібності у складі між геометричним представленням об'єкта та центральною тенденцією кожної групи. Це теоретичне функціонування людського сприйняття та пізнання служить основою для структурного підходу до розпізнавання образів.

Розпізнавання структурних образів, яке іноді називають розпізнаванням синтаксичних образів через його походження в формальній теорії мовлення, спирається на синтаксичні граматики, щоб розрізняти дані різних груп на основі

морфологічних взаємозв'язків (або взаємозв'язків), що містяться в даних. Структурні ознаки, часто називаються примітивними, представляють собою підшаблони (або структурні блоки) та відносини між ними, які складають дані. Семантика, пов'язана з кожною функцією, визначається схемою кодування (тобто вибіркою морфології), що використовується для ідентифікації примітивів у даних. Вектори ознак, породжених структурними системами розпізнавання образів, містять змінюване число ознак (по одній для кожного примітиву, витягнутого з даних), щоб врахувати наявність надлишкових структур, які не впливають на класифікацію. Оскільки взаємовідносини між видобутими примітивами також повинні бути закодовані, векторний компонент повинен включати додаткові функції, що описують відносини між примітивами або приймають альтернативну форму, наприклад, реляційний граф, який може бути проаналізовано синтаксичною граматикою.

Акцент на відносинах між даними робить структурний підхід до розпізнавання образів найбільш розумним для даних, які містять наслідувальну ідентифіковану організацію, таку як дані зображень (які організовуються за місцем розташування всередині візуального рендеринга) та дані про часові ряди (організовані за часом); дані, що складаються з незалежних зразків кількісних вимірювань, не мають упорядкування і вимагають статистичного підходу. Методології, що використовуються для виділення структурних ознак з даних зображення, таких як технології обробки зображення, призводять до таких примітивів, як ребра, криві та регіони; Технологія вилучення функції для даних часової серії включає в себе ланцюгові коди, кусочно-лінійну регресію та фігуру кривої, які використовуються для генерації примітивів, які кодують послідовні, упорядковані за часом співвідношення. Завдання класифікації доходить до ідентифікації, використовуючи синтаксичний аналіз: вилучені структурні ознаки ідентифікуються як представники певної групи, якщо вони можуть бути успішно проаналізовані синтаксичною граматикою. При розрізненні більш ніж двох груп синтаксична граматика необхідна для кожної групи.

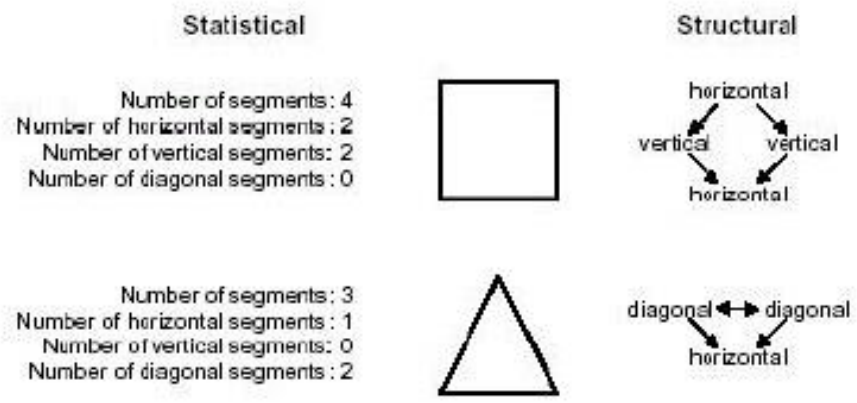


Рисунок 1.4 - Статистичні та структурні підходи до розпізнавання образів, що застосовуються до загальної проблеми ідентифікації. [5]

На рисунку 1.4 зображено, як обидва підходи можуть бути застосовані до однієї проблеми ідентифікації. Мета полягає в тому, щоб диференціювати квадрат і трикутник. Статистичний підхід витягує кількісні характеристики, такі як кількість горизонтальних, вертикальних та діагональних сегментів, які потім передаються класифікатору, що вирішує, до якого класу віднести виділені ознаки. Структурний підхід виділяє морфологічні особливості та їх взаємозв'язки в межах кожної фігури. Використовуючи прямолінійний сегмент як елементарну морфологію, створюється реляційний граф і класифікується за допомогою визначення синтаксичної граматики, яка може успішно проаналізувати реляційний граф. У цьому прикладі як статистичні, так і структурні підходи зможуть точно розрізняти дві геометрії. Однак у більш складних даних дискримінаційність безпосередньо впливає на особливий підхід, використовуваний для розпізнавання образів, оскільки отримані ознаки представляють різні характеристики даних.

Суттєвими відмінностями між статистичними та структурними підходами є: (1) опис, сформований статистичним підходом, є кількісним, тоді як структурний підхід створює опис, що складається з підшаблонів або будівельних блоків; і (2) статистичний підхід розрізняє, ґрунтуючись на численних відмінностях між ознаками різних груп, тоді як граматики використовуються структурним підходом для визначення мови, що охоплює прийнятні конфігурації примітивів для кожної групи. Гібридні системи можуть поєднувати два підходи як спосіб компенсувати недоліки

кожного підходу, зберігаючи при цьому переваги кожного. Як система одного рівня, структурні ознаки можуть використовуватися як зі статистичними, так і з структурними класифікаторами. Статистичні ознаки не можуть бути використані з структурним класифікатором, оскільки вони не мають реляційної інформації, однак статистична інформація може бути пов'язана з структурними примітивами і використовується для вирішення невизначеностей під час класифікації (наприклад, як при розборі з приписаними граматиками) або безпосередньо вкладені безпосередньо в самий класифікатор (наприклад, як при розборі з стохастичними граматиками). Гібридні системи також можуть об'єднувати два підходи в багаторівневу систему, використовуючи паралельну або ієрархічну композицію.

1.2.3 Системи розпізнавання образів

У системі розпізнавання образів виділяють три різні операції: попередньої обробки, вилучення властивостей та класифікацію. Щоб зрозуміти проблему проектування такої системи, ми повинні розуміти проблеми, які необхідно вирішити кожному з цих компонентів [3].

Датчик. Вхід системи розпізнавання образів часто є свого роду перетворювачем, таким як камера або мікрофон. Складність проблеми може цілком залежати від характеристик і обмежень датчика - її пропускної здатності, роздільної здатності, чутливості, спотворення, співвідношення сигнал / шум, затримки тощо.

Сегментація та групування. На практиці предмети, які необхідно розпізнати, часто перетинаються, і система повинна буде визначити, де закінчується один предмет, а де наступний - індивідуальні зразки повинні бути сегментовані. Якщо ми вже виділили цей предмет, то було б простіше сегментувати його зображення. Як ми можемо сегментувати зображення, перш ніж вони будуть класифіковані, або класифікувати їх, перш ніж вони будуть сегментовані? Потрібен спосіб дізнатися, коли ми перейшли з однієї моделі в іншу, або знати, коли ми просто маємо фонову категорію або не маємо жодної категорії. Сегментація є однією з найглибших проблем

розпізнавання образів. Трудно пов'язана з проблемою сегментації - це проблема розпізнавання або групування різних пасток композитного об'єкта.

Виділення ознак. Концептуальна межа між виділенням ознак та правильною класифікацією є дещо умовною: ідеальна функція виділення ознак дає представлення, яке робить роботу класифікатора тривіальною; навпаки, всемогутній класифікатор не потребує допомоги витонченої функції виділення ознак. Різниця виникає з практичних, а не з теоретичних причин.

Традиційна мета виділення ознак полягає в тому, щоб характеризувати об'єкт, який слід визнати вимірами, значення яких дуже схожі для об'єктів однієї категорії, і дуже різні для об'єктів у різних категоріях. Це призводить до ідеї пошуку відмінних функцій, інваріантних до несуттєвих перетворень входу. Загалом, функції, які описують такі властивості, як форма, колір та багато видів текстур, інваріантні для перекладу, обертання та масштабу.

Більш загальна інваріантність буде для обертання щодо довільної лінії в трьох вимірах. Образ навіть такого простого об'єкта, як чашка для кави, зазнає кардинальної зміни, оскільки чашка повертається до довільного кута. Ручка може бути прихована іншою частиною. Крім того, якщо відстань між чашкою та камерою може змінюватися, зображення піддається проекційному спотворенню. Як ми можемо забезпечити, щоб функції були інваріантні для таких складних перетворень? З іншого боку, чи слід визначити різні підкатегорії для зображення чашки та досягти інваріантності обертання при більш високому рівні обробки?

Як і в процесі сегментації, в завданні з вилучення ознак набагато більше проблем - це залежить від домену, ніж є належним класифікацією, і, отже, вимагає знання домену. Хороший класифікатор для сортування риби, ймовірно, буде мати мало користі для ідентифікації відбитків пальців або класифікації мікрофотографії з клітин крові. Проте деякі принципи класифікації моделей можуть бути використані при проектуванні екстрактора властивостей.

1.2.4 Класифікація

Завдання відповідної компоненти класифікатора повної системи полягає в тому, щоб використовувати векторний функціонал, який забезпечує екстрактор функцій, для присвоєння об'єкту категорії. Оскільки досконала класифікація часто неможлива, більш загальним завданням є визначення імовірності кожної з можливих категорій. Абстракція, що забезпечується функціонально-векторним представленням вхідних даних, дозволяє розробляти в основному незалежну від домену теорію класифікації.

Ступінь складності проблеми класифікації залежить від мінливості значень об'єктів для об'єктів у тій же категорії відносно різниці значень об'єктів для об'єктів у різних категоріях. Варіанти значень об'єктів для об'єктів у тій самій категорії можуть бути пов'язані із складністю і можуть бути пов'язані із шумом. Ми визначаємо шум у дуже загальних термінах: будь-яка властивість чутливого шаблону, який не пов'язаний з істинною базовою моделлю, а навпаки, випадковості в світі або сенсорами. Усі нетривіальні рішення та проблеми розпізнавання образів пов'язані з шумом у певній формі.

Одна з проблем, яка виникає на практиці, полягає в тому, що не завжди можливо визначити значення всіх ознак для певних вхідних даних. Як це повинен компенсувати категоризатор? Оскільки наша функція розпізнавання з двома функціями ніколи не мала значення критерію x^*, що змінювалось з однією мінливістю, визначене в очікуванні можливої відсутності функції, як він приймає найкраще рішення, використовуючи тільки наявну функцію? Найвищий спосіб просто припустити, що значення пропущеної функції дорівнює нулю, або середнє значення для вже відомих шаблонів є доказово неоптимальним. Аналогічно, як ми повинні тренувати класифікатор або використовувати його, коли деякі функції відсутні?

1.2.5 Післяобробка

Класифікатор рідко існує у вакуумі. Замість цього він, як правило, повинен використовуватися для того, щоб рекомендувати дії, кожна дія має пов'язану вартість.

Постпроцесор використовує вихід класифікатора, щоб прийняти рішення про рекомендовану дію.

Концептуально, найпростішою мірою ефективності класифікатора є рівень помилки класифікації - відсоток нових шаблонів, які призначені до неправильної категорії. Таким чином, треба шукати класифікацію з мінімальною похибкою. Проте, може бути набагато краще рекомендувати дії, які мінімізують загальну очікувану вартість, яка називається ризиком. Як ми можемо включити знання про витрати і як це вплине на наше рішення щодо класифікації? Чи можемо ми оцінити загальний ризик і, таким чином, сказати, коли наш класифікатор прийнятний ще до того, як ми його виведемо? Чи можемо ми оцінити найнижчий ризик будь-якого класифікатора; щоб побачити, наскільки близько ми зустрічаємо цей ідеал, чи проблема просто занадто важка взагалі?[5]

1.3 Нейронні мережі для задачі розпізнавання

Одним із способів представлення класифікатора для машинного навчання є нейронні мережі.

Спочатку область нейронних мереж була натхненна ціллю моделювання біологічних нейронних систем, але з тих пір вона розійшлася і стала питанням інженерії та досягнення хороших результатів у завданнях з машинного навчання.

Основною обчислювальною одиницею мозку є нейрон. Приблизно 86 мільярдів нейронів можна знайти в нервовій системі людини, і вони пов'язані з приблизно 10^{14} - 10^{15} синапсами. На рисунку 1.5 показано схему біологічного нейрона і на рисунку 1.6 представлена загальна математична модель. Кожен нейрон отримує вхідні сигнали від своїх дендритів і виробляє вихідні сигнали вздовж його (єдиного) аксона. Аксон з часом розгалужується і з'єднується через синапси з дендритами інших нейронів. У розрахунковій моделі нейрона сигнали, які рухаються уздовж аксонів (наприклад, x_0), мультиплікативно взаємодіють (наприклад, w_0x_0) з дендритами іншого нейрону, виходячи з синаптичної сили на цьому синапсі (наприклад, w_0). Ідея полягає в тому, що синаптичні сили (ваги w) можуть навчатися і контролюють силу впливу (і його

напрям: збудливий (позитивний вага) або інгібіторний (негативний вага)) одного нейрона на інший. У базовій моделі дендрити несуть сигнал до тіла комірки, де всі вони отримують підсумки. Якщо остаточна сума перевищує певний поріг, нейрон може спрацювати, посилаючи імпульс уздовж його аксона. У розрахунковій моделі ми припускаємо, що точні таймінги імпульсів не мають значення, і що тільки частота спрацьовування передає інформацію. Виходячи з цієї частотної інтерпретації коду, ми моделюємо частоту спрацювання нейрона *як активаційну функцію f*, яка відображає частоту імпульсів вздовж аксона. Історично загальним вибором функції активації є сигмоподібна функція σ, оскільки вона приймає справжнє значення входу (сила сигналу після суми) і розподіляє її на відстань між 0 і 1.

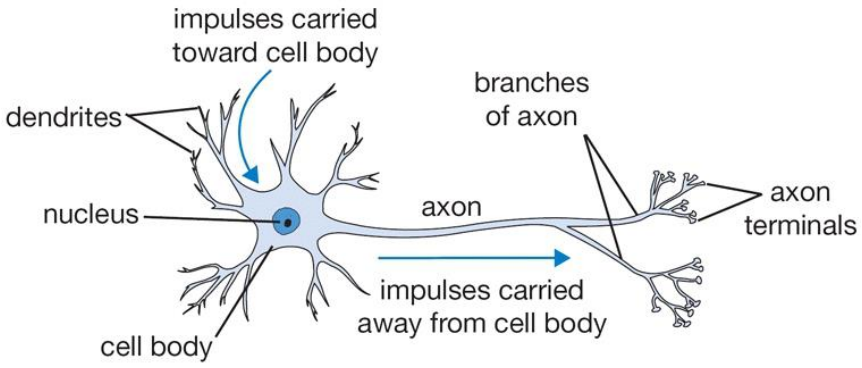


Рисунок 1.5 – Біологічна модель нейрона [7]

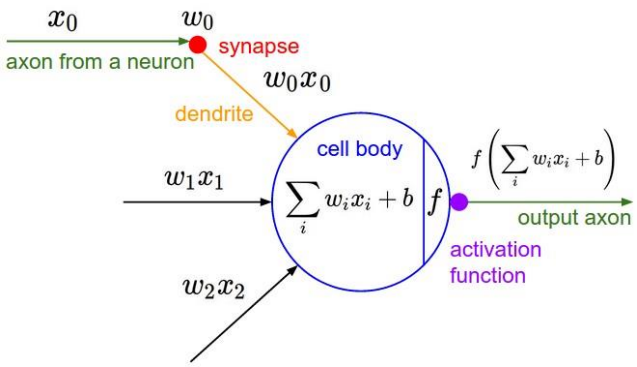


Рисунок 1.6 – Математична модель нейрона [7]

Пряме проходження нейрону можна описати так, як зображено на рисунку 1.7.

```
class Neuron(object):
  # ...
  def forward(self, inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a
number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid
activation function
    return firing_rate
```

Рисунок 1.7 – Код для прямого проходження нейрона [7]

Іншими словами, кожен нейрон виконує скалярне множення його входу із його вагами, додає зміщення та застосовує нелінійність (або функцію активації), в цьому випадку сигмоїду.

1.3.1 Найпопулярніші активаційні функції

Кожна функція активації (або нелінійність) приймає одне число і виконує певну фіксовану математичну операцію на ньому. Існує кілька функцій активації, які ви можете зустріти на практиці:

Sigmoid (Сигмоїда). Сигмоїдна нелінійність представлена у формулі 1.1, а її графік на рисунку 1.8:

$$\sigma(x) = 1/(1 + e^{-x}) \tag{1.1}$$

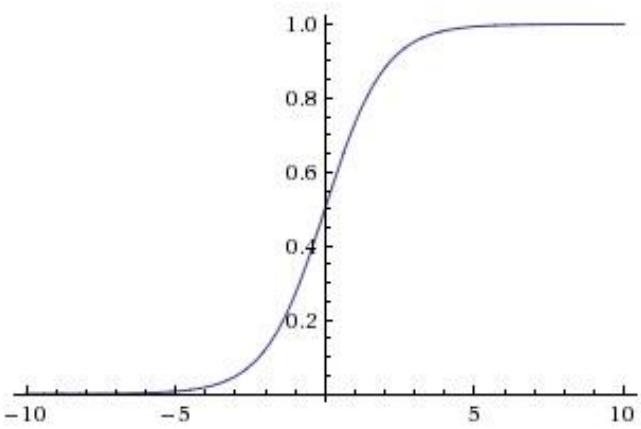


Рисунок 1.8 - Сигмоїдальна нелінійність, яка обмежує дійсні числа до діапазону [0,1] [7]

Вона приймає дійсне число і "стискує" його в діапазон від 0 до 1. Зокрема, великі негативні числа стають 0, а великі позитивні числа стають 1. Сигмоїдна функція часто зустрічається в історичному відношенні оскільки вона має приємну інтерпретацію як швидкість руху нейрона: від не спрацювання зовсім (0) до повністю насиченого спрацювання при прийнятній максимальної частоті (1). На практиці сигмоїдна нелінійність останнім часом випала з користі, і вона рідко використовується. Вона має два основних недоліки:

Сигмоїда насичується і вбиває градієнти. Дуже небажаним властивістю сигмоєвидного нейрону є те, що коли активація нейрона насичується в будь-якому хвості від 0 або 1, градієнт у цих областях майже дорівнює нулю. При зворотному розповсюдженні цей (локальний) градієнт буде помножений на градієнт виходу цього нейрону. Тому, якщо локальний градієнт дуже малий, він фактично «вб'є» градієнт, і практично не буде ніякого сигналу протікати через нейрон до його ваг та рекурсивно до його даних. Окрім того, потрібно приділяти особливу обережність при ініціалізації ваг сигмоєвидних нейронів, щоб запобігти насиченість. Наприклад, якщо початкові ваги занадто великі, більшість нейронів стануть насиченими, і мережа ледь навчиться.

Виходи сигмоїди не нульові. Це небажано, оскільки нейрони у більш пізніх шарах обробки в нейронній мережі будуть отримувати дані, які не нульові. Це має наслідки для динаміки під час градієнтного спуску, тому що якщо дані, що надходять в нейрон, завжди позитивні (наприклад, $x> 0$ елементарно в $f = w^Tx + b$)), то градієнт на вагах w при зворотному поширенні стане будь-яким з усіх позитивний або усіх негативних (залежно від градієнта всього виразу f). Після того, як ці градієнти будуть додані до частини даних, остаточне оновлення для ваг може мати змінні ознаки, що трохи пом'якшує цю проблему. Тому це є незручно, але має менш серйозні наслідки в порівнянні з насиченою проблемою активації вище.

Tanh (Тангенсоїда, тангенційна нелінійність). Ця функція обмежує дійсне число до діапазону [-1, 1]. Подібно сигмоєвидному нейрону, його активація насичується, але на відміну від сигмовидного нейрона його вихід є відцентрованим відносно нуля. Тому на практиці віддають перевагу тангенційній нелінійності, ніж сигмоподібній

нелінійності. Тангенційний нейрон - це масштабований сигмоподібний нейрон, його функція представлена у формулі 1.2, а графік зображено на рисунку 1.9:

$$tanh(x) = 2\sigma(2x) - 1 \tag{1.2}$$

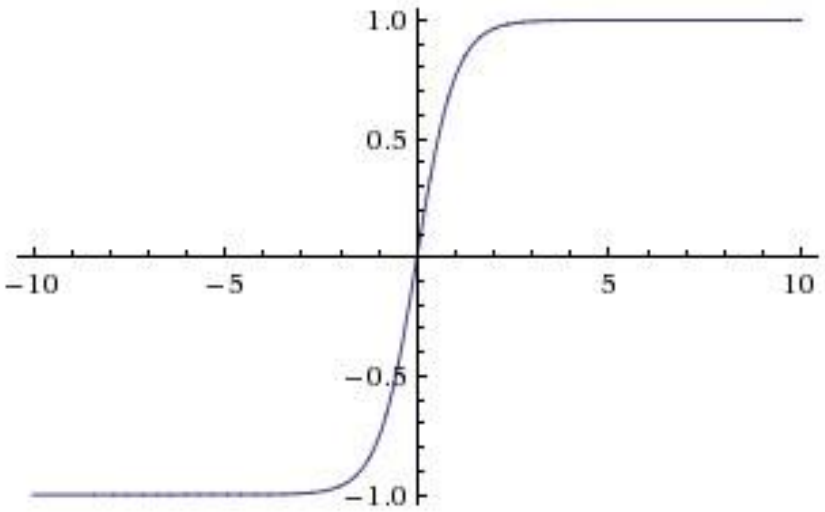


Рисунок 1.9 - Тангенційна нелінійність, яка обмежує дійсні числа до діапазону [-1,1] [7]

ReLU (Rectified Linear Unit). Активаційна функція ReLU стакf дуже популярним протягом останніх кількох років. Її математична формула виглядає так (формула 1.3), а її графік зображено на рисунку 1.10:

$$f(x) = max(0, x) \tag{1.3}$$

Іншими словами, активація – це просто перетин нуля. Є кілька плюсів і мінусів до використання ReLU:

(+) Знайдено значне прискорення збіжності стохастичного градієнтного походження порівняно з функціями Sigmoid / Tanh. Стверджується, що це пов'язано з його лінійною, не насичуючою формою.

(+) У порівнянні з Tanh / Sigmoid нейронами, що передбачають дорогі операції (експоненти тощо), *ReLU* може бути реалізований шляхом простого перетинання матриці активації нуля.

(-) На жаль, нейрони ReLU можуть бути нестабільними під час тренувань і можуть «вмирати». Наприклад, великий градієнт, що протікає через нейрон ReLU, може призвести до оновлення ваг таким чином, що нейрон більше ніколи не активується на будь-яких даних. Якщо це станеться, градієнт, що протікає через нейрон, назавжди стане нульовим з цієї точки. Тобто нейрони ReLU можуть незворотно вмирати під час тренувань. Наприклад, ви можете виявити, що до 40% вашої мережі може бути "мертвою" (тобто нейронами, які ніколи не активуються по всьому навчальному набору даних), якщо швидкість навчання встановлена занадто висока. При правильному налаштуванні швидкості навчання це рідше є проблемою.

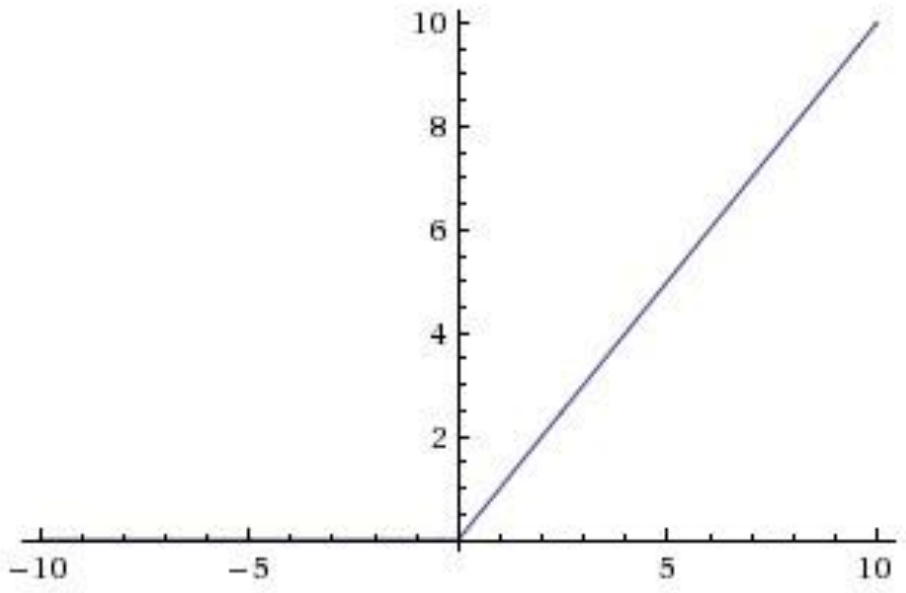


Рисунок 1.10 - Активаційна функція Left: Rectified Linear Unit (ReLU), яка приймає значення 0, коли x < 0 і є лінійно зростаючою, коли x > 0 [7]

1.3.2 Архітектура нейронних мереж

Шарова організація. Нейронні мережі – це нейрони, що об’єднані у граф. Нейронні мережі моделюються як набір нейронів, які пов'язані в ациклічному графі. Іншими словами, виходи деяких нейронів можуть стати входи інших нейронів. Цикли не дозволяються, оскільки це означатиме нескінченний цикл у прямому проході мережі. Моделі нейронної мережі часто організовуються в різні шари нейронів. Для звичайних нейронних мереж найпоширенішим типом шару є повністю з'єднаний шар, в якому нейрони між двома сусідніми шарами повністю попарно пов'язані, але нейрони в одному шарі не мають спільних зв'язків. На рисунках 1.11 і 1.12 наведено два приклади топологій нейронної мережі, які використовують повнозв’язні шари:

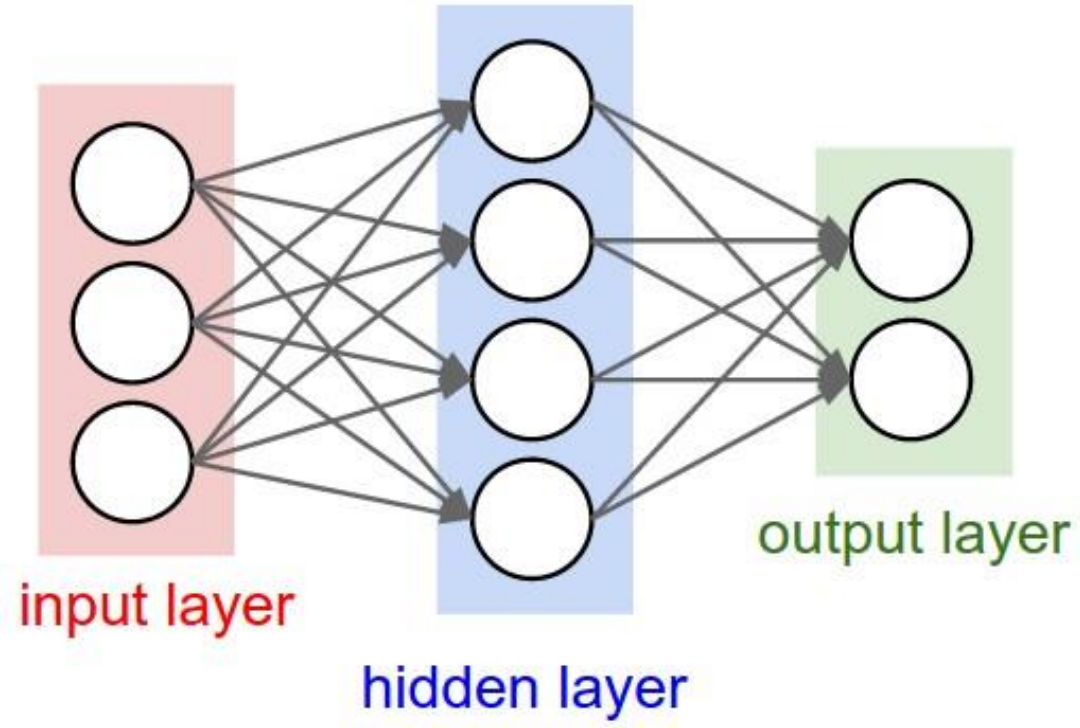


Рисунок 1.11 - Двошарова нейронна мережа (один прихований шар з чотирма нейронами і один вихідний шар з двома нейронами), та три входи [7]

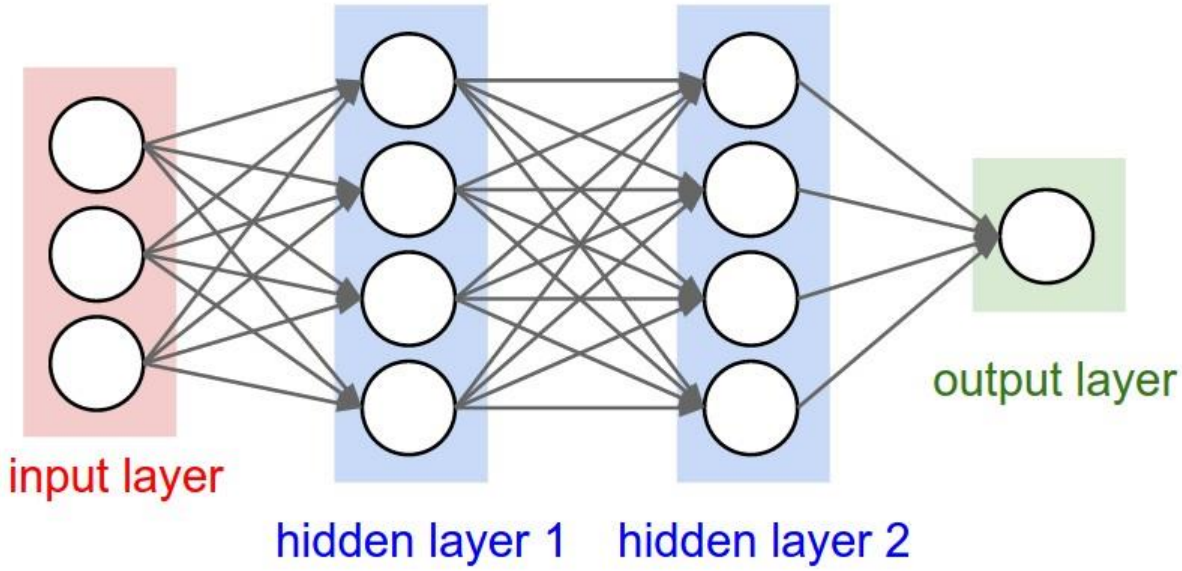


Рисунок 1.12 - Тришарова нейронна мережа з двома прихованими шарами, 3 чотирма нейронами кожна, та одним вихідним шаром [7]

Вихідний шар. На відміну від всіх шарів нейронної мережі, найчастіше нейрони вихідного шару не мають функції активації (або іншими словами, що вони мають лінійну функцію активації ідентичності). Це пояснюється тим, що останній вихідний рівень зазвичай приймається для представлення класів (наприклад, за класифікацією), які є довільними дійсними числами, або певною ціллю дійсного значення (наприклад, у регресії).

Пряме проходження нейронної мережі (feed-forward computation). Повторне множення матриць переплітається з функцією активації. Однією з основних причин того, що нейронні мережі організовані в шари, полягає в тому, що ця структура робить дуже простим та ефективним оцінювання нейронних мереж за допомогою матричних векторних операцій. Працюючи з прикладом тришарової нейронної мережі на схемі вище, вхід буде вектором [3x1]. Всі ваги зв’язків для шару можна зберігати в одній матриці. Наприклад, перша вага прихованого шару *W1* буде мати розмір [4x3], а зміщення для всіх нейронів будуть у векторі *b1* розміру [4x1]. Тут кожен окремий нейрон має ваги в рядку *W1*, тому множення матричного вектора *np.dot (W1, x)* оцінює активацію всіх нейронів у цьому шарі. Аналогічно, *W2* буде матрицею [4x4], яка зберігає зв'язки другого прихованого шару, а *W3* - [1x4] матрицю для останнього

(вихідного) шару. Повне пряме проходження цієї тришарової нейронної мережі - це просто три матричних множення, переплетені із застосуванням функції активації, що зображені на рисунку 1.13:

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations
(4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations
(4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Рисунок 1.13 – Пряме проходження тришарової нейронної мережі

У наведеному вище коді *W1, W2, W3, b1, b2, b3* - це параметри, є навчальними параметрами мережі. Замість одиничного вектора-стовпчика входу, *x* може містити цілу набір тренувальних даних (де кожен приклад буде колонкою з *x*), а потім всі приклади будуть ефективно оцінюватися паралельно. Зверніть увагу, що останній шар нейронної мережі, як правило, не має функції активації (наприклад, він являє собою (реально оцінену) оцінку класу в установці класифікації).

Один із способів розглянути нейронні мережі з повнозв'язнимими шарами полягає в тому, що вони визначають сукупність функцій, параметризованих вагами мережі.

Нейронні мережі, що містять принаймні один прихований шар - універсальні апроксиматори. Тобто, це може бути показано, що за умови неперервної функції $f(x)$ та деякого $\varepsilon > 0$ існує нейронна мережа $g(x)$ з одним прихованим шаром (з розумним вибором нелінійності, наприклад сигмовидної), такими, що $\forall x, |f(x) - g(x)| < \varepsilon$. Іншими словами, нейронна мережа може наближати будь-яку неперервну функцію.

Якщо одного прихованого шару достатньо, щоб наблизити будь-яку функцію, чому використовують більше шарів? Відповідь полягає в тому, що той факт, що двошарова нейронна мережа є універсальним апроксиматором, при цьому є відносно

слабким і не корисним твердженням на практиці. В одному вимірі функція, що зображена у формулі 1.4:

$$g(x) = \sum_i c_i 1(a_i < x < b_i) \quad (1.4)$$

де a, b, c є векторами параметрів,

є також універсальним апроксиматором, але ніхто не вважає за потрібне використовувати цю функцію в машинному навчанні. Нейронні мережі добре працюють на практиці, тому що вони компактно виражають гладкі функції, які добре відповідають статистичним властивостям даних, які ми зустрічаємо на практиці, а також легко навчати за допомогою наших алгоритмів оптимізації (наприклад, градієнтного спуску). Аналогічним чином, той факт, що глибші мережі (з декількома прихованими шарами) можуть працювати краще, ніж мережі з одним-прихованим шаром, є емпіричним спостереженням, незважаючи на те, що їхня представницька сила рівна.

Як на відміну, на практиці часто буває так, що 3-шарові нейронні мережі будуть перевищувати двошарові мережі, проте створення більш глибоких нейронних мереж (4, 5, 6 шарів) рідко допомагає набагато більше. Це суттєво контрастує зі згортковими нейронними мережами (Convolutional Neural Networks, CNN), де визначено, що глибина є надзвичайно важливим компонентом для гарної системи розпізнавання (наприклад, по порядку 10 навчальних шарів). Один аргумент для цього спостереження полягає в тому, що зображення містять ієрархічну структуру (наприклад, обличчя складені з очей, які складаються з країв та ін.), Тому кілька рівнів обробки створюють інтуїтивний сенс для даного домену даних.

Встановлення кількості шарів та їх розмірів. Як вирішити, яку архітектуру використовувати, коли необхідно розв’язати проблему на практиці? По-перше, при збільшенні розміру та кількості шарів в нейронній мережі, потужність мережі збільшується. Тобто, простір представлених функцій зростає, оскільки нейрони можуть співпрацювати, щоб виразити багато різних функцій. Наприклад,

припустимо, що ми мали бінарну проблему класифікації в двох вимірах. Ми могли б тренувати три окремі нейронні мережі, кожен з яких має один прихований шар деякого розміру і отримати наступні класифікатори (рисунок 1.14):

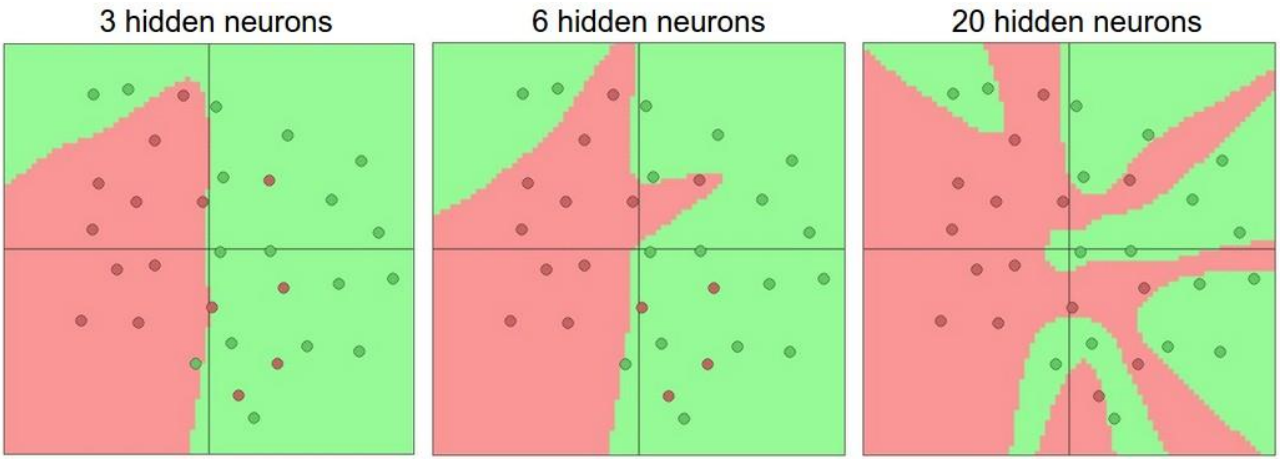


Рисунок 1.14 – Результати тренування мережі з 3, 6 та 20 нейронами у прихованому шарі [7]

На наведеній вище схемі видно, що нейронні мережі з більшою кількістю нейронів можуть виразити більш складні функції. Проте це одночасно і добре (оскільки ми можемо навчитися класифікувати більш складні дані) і погано (оскільки простіше перенавчитись навчальним даним). Перенавчання відбувається, коли модель з високою ємністю підходить для шуму в даних, а не (як передбачається) взаємозв’язків між даними. Наприклад, модель з 20 прихованими нейронами підходить для всіх навчальних даних, але за ціною сегментації простору може неправильно розподіляти червоні та зелені сегменти. Модель з 3 прихованими нейронами має лише представницьку здатність класифікувати дані. Вона моделює дані у вигляді двох блоків і інтерпретує декілька червоних точок усередині зеленого кластера як викиди (шум). На практиці це може призвести до кращого узагальнення на тестовому наборі.

Перевага менших нейронних мереж може бути кращою, якщо дані недостатньо складні для запобігання перенавчання. Проте це неправильно - існує безліч інших переважних способів запобігання перенавчання в нейронних мережах. На практиці,

завжди слід краще використовувати ці методи для контролю перенавчання, а не за кількістю нейронів.

Точна причина полягає в тому, що менші мережі важче тренувати за допомогою таких локальних методів, як градієнтний спуск: зрозуміло, що їх функції втрат мають порівняно небагато локальних мінімумів, але виявляється, що багато з цих мінімумів легше зближаються, і що вони погані (тобто з великою втратою). І навпаки, більші нейронні мережі містять значно більше локальних мінімумів, але ці мінімуми виявилися набагато кращими з точки зору їх фактичної втрати. Якщо ви тренуєте маленьку мережу, то остаточна втрата може відображати гарну кількість дисперсії - у деяких випадках вам пощастить і сходити в хорошому місці, але в деяких випадках ви потраппляєте в один з поганих мінімумів. З іншого боку, якщо ви тренуєте велику мережу, ви почнете знаходити багато різних рішень, але різниця в остаточному досягненні втрат буде набагато меншою. Інакше кажучи, всі рішення приблизно однаково добре, і менше покладаються на успіх випадкової ініціалізації.

Сила регуляризації - це найкращий спосіб контролювати перенавчання нейронної мережі. Можна оцінити результати, що були досягнуті за трьома різними налаштуваннями:

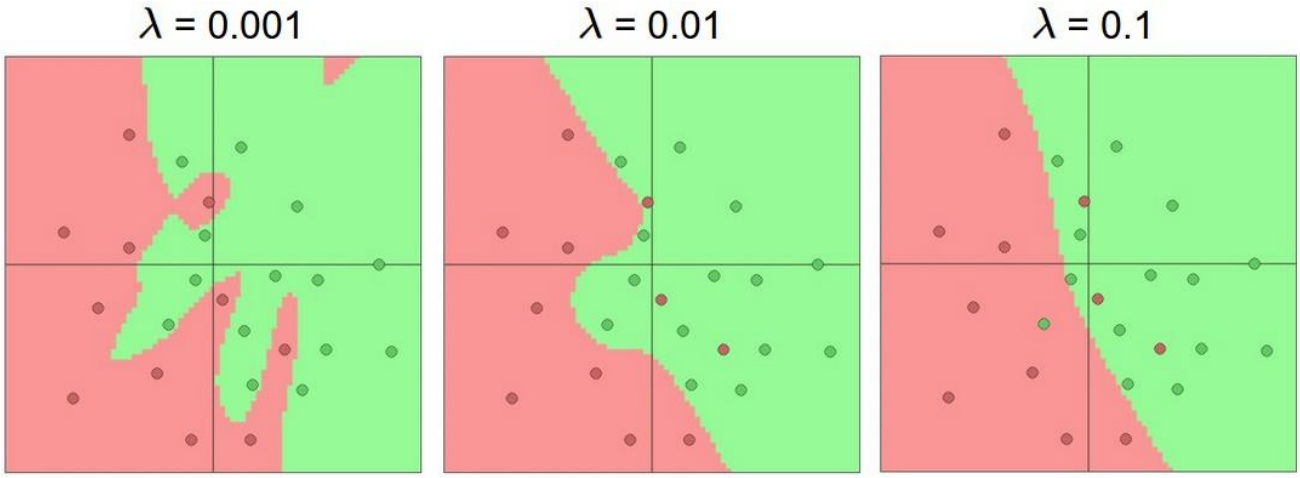


Рисунок 1.15 – Результати роботи нейронної мережі із різною силою регуляризації

Вихід - це те, що ви не повинні використовувати менші мережі, тому що ви боїтеся перенавчання. Замість цього, ви маєте використовувати як велику нейронну

мережу, як дозволяє ваш обчислювальний бюджет, а також використовувати інші методи регуляризації для контролю за перенавчанням [7].

1.4 Згорткові нейронні мережі

Convolutional neural network (CNN) схожі на звичайні нейронні мережі: вони складаються з нейронів, які мають вагові коефіцієнти та зміщення (рисунок 1.16). Кожен нейрон отримує кілька входів, виконує скалярний твір. Вся мережа як і раніше описує єдину розрізнювальну оціночну функцію: з вихідних пікселів зображення на одному кінці мереж отримується ступінь приналежності до того, чи іншого класу. Також, цей тип мереж має функцію втрат (наприклад, SVM / Softmax) на останньому повнозв’язному шарі.

Головною відмінністю архітектури CNN є припущення про те, що входами є зображення, які дозволяють перетворювати певні особливості цих зображень в архітектуру мережі. Саме це зробило передавальну функцію більш ефективною в реалізації та значно зменшилась кількість параметрів в мережі.

Звичайна нейронна мережа. Нейронні мережі отримують вхід (один вектор), і перетворюють його через ряд прихованих шарів. Кожен прихований шар складається з набору нейронів, де кожен нейрон повністю зв’язаний з іншими нейронами в попередньому шарі, і нейрони, що знаходяться в одному шарі, функціонально незалежні та не мають спільних зв’язків. Останній повнозв’язний шар називається "вихідний шар" і в налаштуваннях класифікації він представляє ступінь приналежності до того, чи іншого класу.

Звичайні нейронні мережі погано масштабуються для великих зображень. У CIFAR-10, зображення мають розмір 32x32x3 (32 в ширину, 32 у висоту, 3 кольорових каналів), так що один повнозв’язний нейрон в першому прихованому шарі звичайної нейронної мережі буде мати 32 * 32 * 3 = 3072 ваг. Ця сума не дуже велике для управління нею, але очевидно, що ця повнозв’язна структура не масштабується для великих зображень. Наприклад, зображення більшого розміру, наприклад 200x200x3, призведе до того, що в нейронній мережі буде 200 * 200 * 3 = 120000 ваг. Така

повнозв’язність є марнотратною і велика кількість параметрів може швидко привести до перенавчання.

Тривимірна розмірність нейронів. Основною перевагою CNN (рисунок 2.3) є те, що входами є зображення, які визначають архітектуру більш правильним шляхом. Зокрема, на відміну від звичайної нейронної мережі, шари в ConvNet мають нейрони розташовані в трьох вимірах: ширина, висота, глибина (рисунок 2.4). Нейрони в шарі будуть з’єднуються тільки з невеликою областю шару перед ним, а не з’єднується по принципу «багато до багатьох». Крім того, кінцевий вихідний шар для CIFAR-10 має розмір 1x1x10, тому що в кінці архітектури CNN повне зображення зменшується в єдиний вектор класу оцінки.

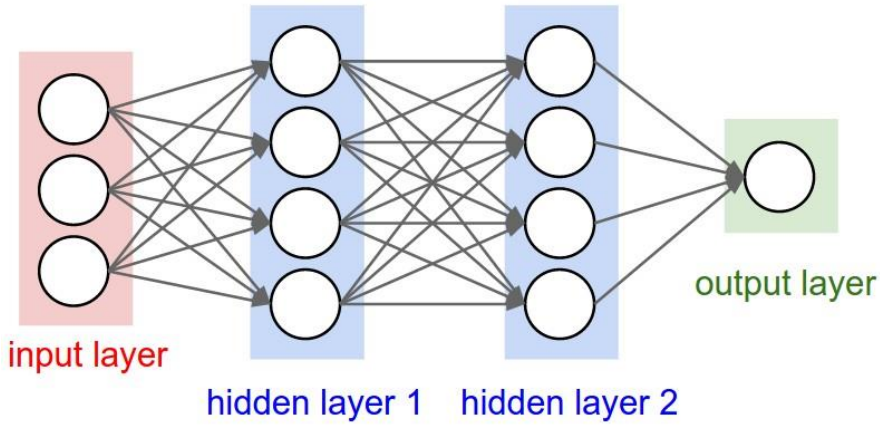


Рисунок 1.16 - Звичайна тришарова нейронна мережа [8]

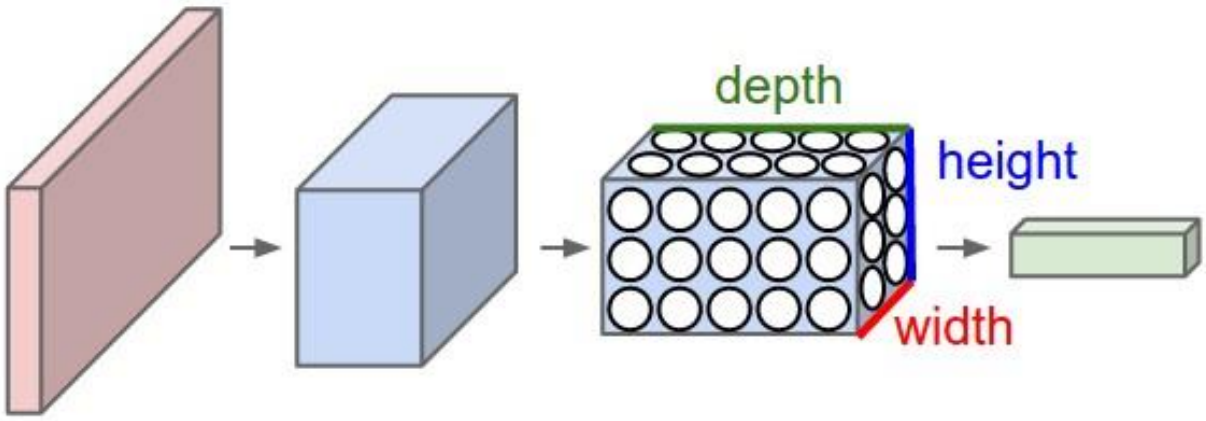


Рисунок 1.17 – Згорткова нейронна мережа [8]

Кожен шар CNN перетворює тривимірний вхід у тривимірний вихід функції активації. На рисунку вище червоний вхідний шар представляє зображення, тому ширина і висота будуть вимірами зображення, а глибина буде рівна 3 (червоний, зелений та синій канали).

Шари, що використовуються для побудови CNN. Кожен шар CNN перетворює одну розмірність активацій в іншу, проходячи через диференціюючу функцію. Зазвичай використовуються три типи шарів для побудови архітектури CNN: convolutional layer (CONV, згортковий шар), pooling layer (POOL, субдискретизуючий шар, тобто шар, який проводить операцію підвибірки), fully-connected layer (FC, повнозв’язний шар, як у звичайній нейронній мережі). Поєднання цих шарів і утворює архітектуру CNN.

Для прикладу розглянемо просту архітектуру мережі для CIFAR-10 набору [INPUT - CONV - RELU – POOL - FC]:

1. INPUT (вхід, рисунок 1.18) [32x32x3] буде містити вихідні значення пікселів зображення, в цьому випадку це ширина 32 пікселів, висота 32 пікселів та глибина, яка рівна трьом по кількості колірних каналів;

2. CONV шар (згортковий шар) буде обчислювати вихід нейронів, які з'єднані з локальними областями на вході, де кожне обчислення полягає в скалярному творі між вагами та областю входу, з якою вони з’єднані. Результуюча розмірність буде [32x32x12].

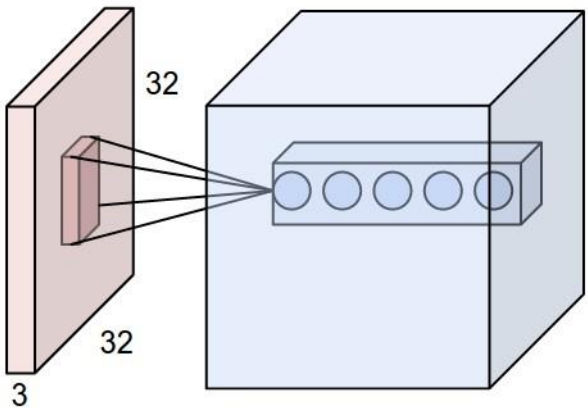


Рисунок 1.18 - Приклад розмірності входу (32x32x3) та приклад розмірності нейрону CONV шарі [8]

3. ReLu (Rectified Linear Units) шар поелементно застосовує функцію функцію активації, таку як max(0,x) з порогом в нулі. Розмірність залишається без змін ([32x32x12]).

4. POOL шар буде виконує операцію субдискретизації просторових розмірів (ширина, висота), в результаті чого розмірність стає рівною [16x16x12].

5. FC шар обчислює степінь приналежності до класу, результат буде мати розмірність [1x1x10], де кожен з 10-ти номерів відповідає ступеню приналежності до того чи іншого класу, наприклад, до 10 класів у CIFAR-10. Як і у випадку звичайних нейронних мереж кожен нейрон в цьому шарі буде під'єднаний до всіх нейронів в повному обсязі.

Таким чином, CNN шар за шаром перетворює пікселі початкового зображення у кінцеві значення степеню приналежності до класу (приклад перетворень представлений на рисунку 1.19). Деякі шари містять параметри, а в деяких їх немає. Зокрема, шари CONV та FC виконують перетворення, які є результатом не тільки активацій всього об'єму входу, але також і параметрів (ваг і зсувів нейронів). З іншого боку, ReLu та POOL шари реалізують закріплений за ними функціонал. Параметри в шарах CONV та FC навчаються за допомогою методу градієнтного спуску, так степінь приналежності до класу, що CNN обчислює узгоджуються з мітками в навчальному наборі для кожного зображення.

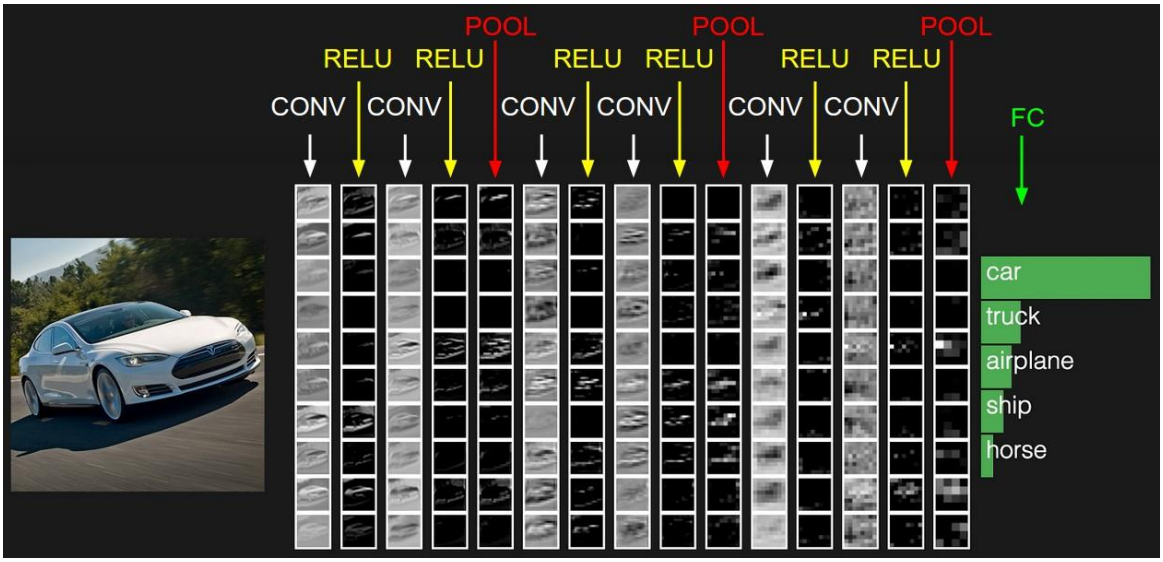


Рисунок 1.19 – Шари згорткової нейронної мережі [8]

Особливості CNN. Першою особливістю є local connectivity (локальне з’єднання). Якщо розмірність входів велика, наприклад, таких як зображення, то недоцільно поєднувати нейрони за принципом «всі до всіх». Замість цього кожен нейрон поєднується тільки з локальною областю входу, тобто лише деякою частиною. Просторова протяжність зв'язку є гіперпараметром, який називається полем сприйняття нейрона.

Backpropagation (зворотне поширення). Зворотний прохід для операції згортки (як для даних і ваг) також є згорткою (але вже для фільтрів).

Parameter sharing (спільне використання параметрів). Ця особливість використовується в CONV шарі, щоб управляти кількістю параметрів. Можливо значно зменшити кількість параметрів, зробивши таке припущення: якщо одну особливість треба було обчислити при деякому просторовому положенні (x, y), то воно також може бути обчислено в іншому місці (x2, y2). На практиці під час зворотного поширення, кожен нейрон в буде обчислювати градієнт для його ваги, але ці градієнти будуть додаватися під час проходження через кожен зріз по глибині (який буде двовимірним) і оновлювати тільки один набір ваг на зріз за один раз.

Всі нейрони в одному зрізі по глибині використовують один і той же вектор ваги, тому при проходженні CONV шару кожен зріз по глибині може бути обчислений як згортка ваг нейронів. Таким чином, цей вектор є загальним для набору ваг в якості фільтра (або ядра), який є згорнутим з входом. Результатом цієї згортки є карта activation map (карта особливостей).[8]

1.5 Висновки за розділом

Машинне навчання дозволяє розв’язувати задачі, які важко розв’язати за допомогою конкретного алгоритму, де необхідно постійно адаптуватись до нових, непередбачених входів. До такого класу задач належать і задача розпізнавання – віднесення об’єкту до певного відомого класу. Саме розпізнавання поділяється на кілька підзадач: виділення об’єкту, виділення ознак на ньому, віднесення до певного класу. В якості класифікатора, що відносить виділений об’єкт до певного класу, може

використовуватись нейронна мережа. Вона представляє собою математичну модель біологічних нейронних систем, що складаються з нейронів та зв’язків між ними. Ідея цієї моделі полягає в тому, що після проведення навчання (налаштування ваг нейронів), така модель зможе працювати з новими, раніше невідомими, даними. Для розпізнавання зображень найчастіше використовують згорткову нейронну мережу, оскільки її головною особливістю є тривимірна розмірність нейронів, що зменшує загальну кількість зв’язків між усіма нейронами (не всі шари мають бути повнозв’язними). Завдяки цьому ця мережа краще працює саме з такими ієрархічними даними, як зображення (тобто стає можливим виділення додаткових ознак з уже виділених ознак).

2 ВИБІР БІБЛІОТЕК МАШИННОГО НАВЧАННЯ

2.1 Імперативний та символічний підходи

Різниця між імперативними (imperative) і символічними (symbolic) фреймворками полягає у способі виконання операцій. Імперативний підхід виконує операцію саме в той час, коли її викликають. Це означає, що коли виконується рядок $c = b * a$ з лістингу на рисунку 2.1, то програма виконує обчислення значень, що були вказані вище.

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
```

Рисунок 2.1 - Лістинг імперативного підходу програмування

Символічний підхід відрізняється тим, що спочатку визначається функція, яка буде виконуватись проте без конкретних значень (рисунок 2.2). Ця функція має змінні, які потім будуть ініціалізуватись вже реальними значеннями. Тоді необхідно скомпілювати функцію, та виконати її вже із реальними значеннями.

```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```

Рисунок 2.2 - Лістинг символічного підходу програмування

Під час виконання рядку *c = b * a* ніяких обчислень не відбувається. Замість цього ця операція генерує граф обчислень (computation graph), який представляє це обчислення. Приклад графу, який буде побудовано під час ви-конання рядка *c = b * a* представлено на рисунку 2.3.

Важливим моментом символічному підходу є те, що програмо явно або неявно містить крок компіляції. Компіляція перетворює граф обчислень в функцію, яка буде виконана пізніше.

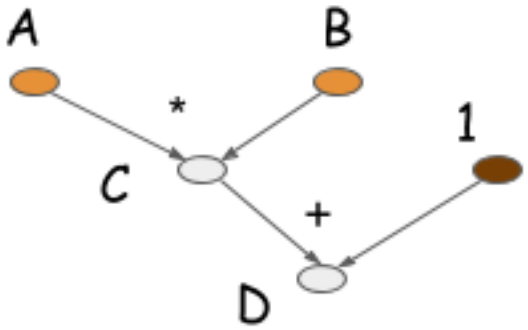


Рисунок 2.3 - Граф обчислень, згенерований програмою [9]

У лістингу, що зображено на рисунку 2.2 фактичне обчислення відбудеться під час виконання останнього рядку. Символічний підхід визначає чітке роз-ділення між створенням графа обчислень і його виконання. Для нейронних мереж зазвичай усю модель визначають як єдиний граф обчислень.

Імперативний підхід більш інтуїтивно зрозумілий, гнучкий. Будь-який алгоритм з імперативним підходом пишеться «as is», майже природньою мовою. Символічний підхід є більш ефективним за пам’яттю та швидкістю. Для виконання обчислення «тут і зараз» необхідно виділяти пам’ять під кожний рядок програми. Коли ми будуємо граф обчислень і знаємо, що в результаті нам потрібно тільки значення d з рисунку 4, то стає можливим перевикористовувати пам’ять, що була виділена для зберігання проміжних значень. Для імперативного підходу ми можемо очищати пам’ять, наприклад видаляти з пам’яті біти, виділені для b, щоб зберегти c, потім виділити з пам’яті c, щоб зберегти d.

Символічний підхід є більш обмеженим. Коли відбувається виклик компіляції графу обчислень, то система знає, що тільки значення d, є необхідним. Проміжні значення є невидимими для нас.

Символічний підхід може безпечно перевикористати пам’ять, яку було виділено для обчислень «тут і зараз». Проте при цьому ми втрачаємо доступ до c, тому імперативний підхід тут має більшу перевагу, оскільки ми можемо отримати значення будь-якої змінної у потрібний для нас час.

Символічний підхід може використовувати інший спосіб оптимізації, який називається складання (folding) (рисунок 2.4). Це означає, що, наприклад, операція множення і додавання може бути об’єднана в одну операцію. Якщо обчислення проводяться на GPU, то тільки одне ядро GPU буде використано для цієї операції замість двох. Така оптимізація збільшує швидкість обчислень.

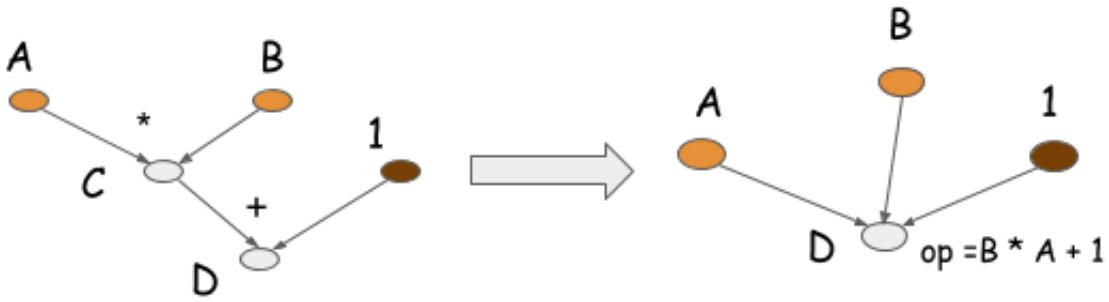


Рисунок 2.4 - Операція складання [9]

Операція складання неможлива для імперативного підходу, оскільки проміжні значення можуть бути використані в майбутньому. Операція складання можлива для символічного підходу, оскільки ми отримуємо повний граф обчислень та чітко розуміємо, які значення нам потрібні, а які ні [9].

2.2 Короткий огляд бібліотек, що були обрані для порівняння

Для порівняння було обрано 5 найпопулярніших бібліотек машинного навчання: TensorFlow [10], MXNet [11], CNTK [12], PyTorch [13] і Caffe [14].

2.2.1 TensorFlow

TensorFlow - це система машинного навчання, яка працює у широкомасштабних та в неоднорідних середовищах. TensorFlow використовує графи потоку даних (dataflow graphs) для представлення обчислень, спільного стану та операцій, які змінюють цей стан. TensorFlow дозволяє розробникам експериментувати з новими алгоритмами оптимізації та навчання. TensorFlow підтримує безліч додатків зосереджуючись на тренуванні та висновку щодо роботи глибоких нейронних мереж. Кілька служб Google використовують TensorFlow у виробництві, він випущений як проект із відкритим кодом, і він широко використовується для дослідження машинного навчання. Приклад класифікатора зображень, написаного на TensorFlow зображено на рисунку 2.5.

```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```

Рисунок 2.5 - Класифікатор зображень, написаний на TensorFlow [10]

TensorFlow забезпечує просте програмування на основі абстракції потоку даних, яка дозволяє користувачам розгортати додатки на розподілених кластерах, локальних робочих станціях, мобільних пристроях і спеціально розроблених прискорювачів. Високорівневий інтерфейс для написання скриптів обгортає

конструювання графів потоку даних і дозволяє користувачам експериментувати з різними архітектурами моделі та алгоритмами оптимізації не змінюючи основну систему.

Основні принципи проектування TensorFlow:

Графи потоку даних примітивних операторів (Dataflow graphs of primitive operators). TensorFlow використовує представлення потоку даних для його моделі, яка являє собою індивідуальні математичні оператори (наприклад, множення матриць, згортка тощо) як вузли в графі потоку даних. Цей підхід полегшує створення користувачами нових шарів, використовуючи високорівневий інтерфейс для написання скриптів. На додаток до функціональних операторів, TensorFlow має змінний стан та операції, що оновлюють його, як вузли на графу потоку даних, що дозволяє експериментувати з різними правилами оновлення.

Відкладене виконання (Deferred execution). Типова програма TensorFlow має дві чіткі фази: перша фаза визначає програму (наприклад, нейронну мережу, яку слід навчати, та правила оновлення) як символічний граф потоку даних із змінними, які будуть заповнюватись вхідними даними та змінними, що представляють стан; і друга фаза виконує оптимізовану версію програми на наборі доступних пристроїв. Відклавши виконання, поки не буде доступна вся програма, TensorFlow може оптимізувати фазу виконання за допомогою глобальної інформації про обчислення. Наприклад, TensorFlow досягає високої утилізації GPU, використовуючи структурну залежність графу, щоб видати послідовність ядер для GPU не чекаючи проміжних результатів.

Загальна абстракція для гетерогенних прискорювачів На додаток до універсальних пристроїв, таких як багатоядерний процесор і графічний процесор, спеціальних прискорювачі для глибокого машинного навчання може досягти значних покращень у продуктивності і енергозбереження. У Google побудовано спеціальний блок обробки тензора (Tensor Processing Unit, TPU) для машинного навчання. Для підтримки цих прискорювачів в TensorFlow, визначено загальну абстракцію для пристроїв. Як мінімум, пристрій повинен реалізувати методи для створення ядра для виконання, розподілу пам'яті для входів і виходів і передачі буферу з і до пам’яті

хоста. Кожен оператор (наприклад, матричне множення) може мати кілька спеціалізованих реалізацій для різних пристроїв. У результаті та ж сама програма може легко націлюватися на графічні процесори, TPU або мобільні процесори як це потрібно для тренувань.

TensorFlow використовує тензори примітивних значень як загальний формат обміну, який розуміють всі пристрої. На найнижчому рівні, всі тензори в TensorFlow щільні. Це рішення гарантує, що на найнижчих рівнях система має просту реалізацію для виділення пам'яті і серіалізації даних, тим самим зменшуючи основні накладні витрати.

Модель виконання TensorFlow. TensorFlow використовує єдиний граф потоку даних для представлення всіх обчислень та стану в алгоритмі машинного навчання, включаючи окремі математичні операції, параметри та їх правила оновлення, а також попередню обробку вхідних даних (рисунок 2.6.).

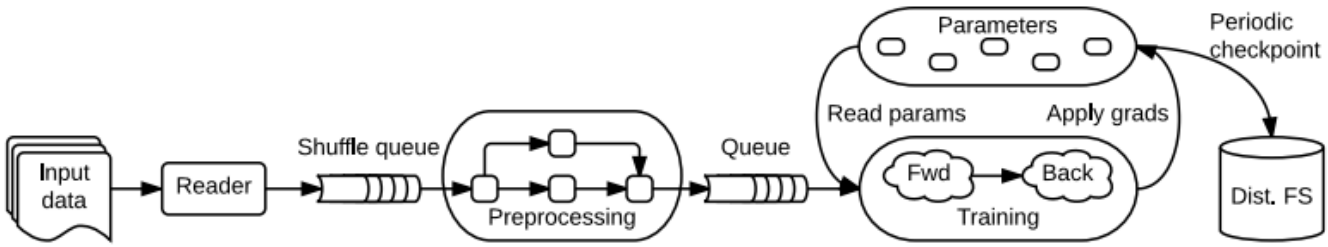


Рисунок 2.6 - Схематичний граф потоку даних TensorFlow для тренування, що містить підграфи для зчитування вхідних даних, попередньої обробки, тренування та періодичне збереження [10]

Граф потоку даних виражає комунікацію між підобчисленнями явно, таким чином, що дозволяє легко виконувати незалежні обчислення паралельно і розділити обчислення на кількох пристроях. TensorFlow відрізняється від пакетних систем потоку даних у двох аспектах:

1. Модель підтримує кілька одночасних виконань на перекритті підграфів загального графа.

2. Індивідуальні вершини можуть мати змінюваний стан, який може бути розподіленим між різними виконаннями графа.

Основне спостереження в архітектурі сервера параметрів полягає в тому, що змінюваний стан має вирішальне значення для тренування дуже великих моделей, тому що стає можливим робити оновлення для дуже великих параметрів на місці та поширювати ці оновлення для паралельних обчислень, настільки швидко, як це можливо. Потік даних із змінним станом дозволяє TensorFlow імітувати функціональність сервера параметрів, але з додатковою гнучкістю, оскільки це стає можливим для виконання довільних підграфів потоків даних на машинах, що приймають загальні параметри моделі. В результаті, користувачі змогли експериментувати з різними алгоритми оптимізації, схеми узгодженості та стратегії розпаралелювання.

Елементи графу потоку даних. У графі TensorFlow кожна вершина являє собою одиницю локального обчислення, і кожне ребро представляє вихід від або вхід до вершини. Розрахунки у вершинах – це операції, а значення, які проходять уздовж ребер – це тензори.

Тензор. У TensorFlow всі дані моделюються як тензори (n-мірні масиви) з елементами, які мають примітивний тип, таких як *int32*, *float32* або *string* (де *string* може бути довільними бінарними даними). Тензори, природно, представляють входи і результати загальних математичних операцій в багатьох алгоритмах машинного навчання: наприклад, множення матриць приймає два 2-D тензори і віддає 2-D тензор; пакет 2-D згортки приймає два тензори 4-D і віддає ще один 4-D тензор.

Операції. Операція приймає $m \geq 0$ тензорів на вхід і віддає $n \geq 0$ тензорів на виході. Операція має іменований "тип" (наприклад, Const, MatMul або Assign) і може мати нуль або більше атрибутів під час компіляції, які визначають її поведінку. Операція може бути поліморфічною і варіативною під час компіляції: її атрибути визначають обидва очікувані типи та арність її входів і виходів.

Наприклад, найпростіша операція *Const* не має вхідних даних і має єдиний вихід; її значення є атрибутом під час компіляції. Наприклад, *AddN* сумує декілька

тензорів того ж примітивного типу, і вона має атрибут типу *T* і цілочисельний атрибут *N*, який визначає його тип.

Операції з відслідковуванням стану: змінні (*variables*). Операції можуть містити змінний стан, що читається і / або записується під час кожного виконання. Операція *Variable* містить змінний буфер, який може бути використаний для зберігання спільних параметрів моделі, яку тренують. *Variable* не має вхідних даних; і віддає посилання обробника (*reference handle*), який діє як типізована можливість читання та запису буфера. Операція Read приймає посилання обробника *r* як вхід і вихід значення змінної *(State [r])* як щільний тензор. Інші операції змінюють буфер: наприклад, *AssignAdd* приймає посилання обробника *r* та значення тензору *x*, а при виконанні оновлює *State'[r]* ← *State[r]* + *x*. Подальші операції *Read(r)* віддають значення *State'[r]*.

Операції з відслідковуванням стану: черги (queues). TensorFlow включає кілька реалізації черги. Найпростіша черга – це *FIFOQueue*, що володіє внутрішньою чергою тензорів, і підтримує одночасний доступ у порядку first-in-first-out. Інші типи черг віддають тензори у випадковому та пріоритетному порядку, які гарантують, що вхідні дані відбираються відповідно. Як *Variable*, операція *FIFOQueue* віддає посилдання обробника, який може бути прийнятий однією з стандартних операцій черги, таких як *Enqueue* і *Dequeue*. *Enqueue* буде блокувати, якщо його задана черга повна, і *Dequeue* блокуватиметься, якщо ця черга порожня. [10]

2.2.2 MXNet

Можливі парадигми програмування варіюються від *імперативних*, де користувач вказує "як" повинно бути виконане обчислення, а також *декларативних*, де користувач фокусується на тому, "що" буде зроблено.

Що стосується проблеми парадигм програмування, то це те, як здійснюється обчислення. Виконання може бути *конкретним*, де результат одразу ж повертається у тому самому потоці, або *асинхронним* чи *відкладеним*, де операції збираються та перетворюються в граф потоку даних як проміжне представлення, перш ніж перейти

до виконання на доступних пристроях. Основні відмінності між імперативним і декларативним підходом зображено у таблиці 2.1.

Таблиця 2.1 – Відмінності між імперативним та символічним підходами

	Імперативний підхід	Декларативний підхід
a = b + 1	Одразу ж виконує та зберігає результати в *a* того ж самого типу як і *b*	Повертає граф обчислень; прив'язує дані до b і виконує обчислення пізніше
Переваги	Концепутально є прямолінійним підходом, і часто працює без проблем з вбудованими структурами даних, функціями, дебагером і сторонніми бібліотеками конкретної мови програмування	Отримує повний граф обчислень перед виконанням, корисний для оптимізації і звільнення пам'яті. Також зручно реалізовувати функції такі як завантаження, збереження і візуалізація

MXNet має на меті комбінацію переваг цих різних підходів. Декларативне програмування пропонує чітку межу на глобальному графі обчислень, відкриваючи більше можливостей для оптимізації, тоді як імперативні програми пропонують більшу гнучкість. В контексті глибокого навчання, декларативне програмування є корисним для визначення структури обчислень в конфігураціях нейронної мережі, тоді як імперативне програмування є більш природним для параметрів оновлення та інтерактивного налагодження.

Незважаючи на підтримку декількох мов та поєднання різних парадигм програмування, MXNet здатний поєднати виконання з єдиним движком на бекенді. Движорк відстежує дані залежностей через обчислювальні графи та імперативні операції, а також ефективно розподіляє їх за часом. MXNet сильно знизили об'єм пам'яті, виконуючи оновлення на місці та перевикористовують пам'ять всюди, де це можливо. MXNet розробили компактний API для того, щоб запустити програму MXNet на кількох машинах з невеликими змінами.

Інтерфейс програмування

Декларативні символічні вирази. MXNet використовує символічні вирази, що мають багато виходів, *Symbol*, що визначають обчислювальний граф. Символи складаються у оператори, такі як прості матричні операції (наприклад, "+"), або складні шари нейронних мереж (наприклад, шар згортки). Оператор може приймати кілька входів, віддавати більше одного вихідного значення і мають внутрішні змінні стану. Змінна може бути або вільна, яку ми можемо прив'язати із значенням пізніше, або з вихідом іншого символу. На рисунку 2.7 показано побудову багатошарової мережі і деяких операцій з шарами нейронної мережі.

```
using MXNet
mlp = @mx.chain mx.Variable(:data) =>
  mx.FullyConnected(num_hidden=64) =>
  mx.Activation(act_type=:relu)    =>
  mx.FullyConnected(num_hidden=10) =>
  mx.Softmax()
```

```
>>> import mxnet as mx
>>> a = mx.nd.ones((2, 3),
... mx.gpu())
>>> print (a * 2).asnumpy()
[[ 2.  2.  2.]
 [ 2.  2.  2.]]
```

Рисунок 2.7 – Багатошарова мережа та операції із шарами за допомогою MXNet [11]

Для виконання символу нам потрібно пов'язати вільні змінні з даними та визначити потрібні результати. Крім виконання ("прямий прохід"), символ підтримує автоматичне символічне диференціювання ("зворотній прохід"). Інші функції, такі як завантаження, збереження, оцінка пам'яті та візуалізація, також надаються для символів.

NDArray: імперативне обчислення тензора. MXNet пропонує *NDArray* з імперативним обчисленням тензора для заповнення прогалини між декларативним символічним виразами та мовою хоста. На рисунку вище наведено приклад, який виконує множення константних матриць на GPU, а потім виводить результати за допомогою *numpy.ndarray*.

NDArray абстракція працює без проблем з виконанням, оголошеними *Symbol*, оскільки MXNet дозволяє змішувати імперативне тензорне обчислення з іншим. Наприклад, дано символічну нейронну мережу та функцію оновлення ваги, наприклад $w = w - \eta g$. Тоді ми зможемо реалізувати градієнтний спуск по

*while (1) {net.foward_backward (); net.w - = eta * net.g};*

Вищезгадана реалізація така ж ефективна, як і реалізація, що використовує єдиний але часто набагато складніший символічний вираз/ Причиною є те, що MXNet використовує ліниву ініціалізацію NDArray та движка, що може правильно вирішувати залежність даних між двома підходами.

Графи обчислень. Прив’язаний символічний вираз представляється у вигляді графу обчислень для виконання. На рисунку 2.8 показано пряме і зворотнt проходження графа. Перед виконанням MXNet перетворює граф, щоб оптимізувати ефективність і виділяє пам’ять для внутрішніх змінних.

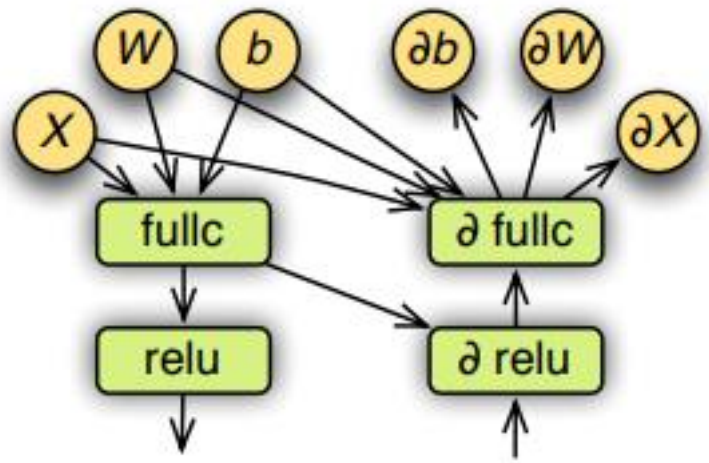


Рисунок 2.8. – Пряме та зворотне проходження графа [11]

Оптимізація графа. Спочатку обраховується тільки необхідний підграф, що потрібен для результатів, отриманих під час прив’язки. Наприклад, при прогнозуванні потрібний лише граф при прямому проходженні, тоді як при виділенні ознак з внутрішніх шарів, останні шари можуть бути пропущені. По-друге, оператори можуть бути згрупован. Нарешті, у MXNet вручну оптимізовано "Великі" операції, такі як шари в нейронній мережі.

Розподіл пам'яті. Час життя кожної змінної, а саме період між створенням і останнім часом використання, відоме графу обчислень. Таким чином стає можливим

повторно використати пам'ять для цих змінних. Однак для ідеального розподілу стратегія вимагає $O(n^2)$ складность часу, де n - це число змінних. [11]

2.2.3 CNTK

CNTK – це бібліотека для опису навчальних машин. Хоча це призначено для нейронних мереж, навчальні машини є довільними, оскільки логіка машини описується серією обчислювальних кроків в обчислювальній мережі. Обчислювальна мережа визначає функцію, яку слід навчати, як орієнтований граф, де кожен кінцевий вузол складається з вхідного значення або параметра, а кожен незалежний вузол представляє матрицю або тензорну операцію для своїх дітей. Перевага CNTK полягає в тому, що, як тільки була описана обчислювальна мережа, всі розрахунки, необхідні для вивчення параметрів мережі, обробляються автоматично. Немає необхідності аналітично виводити градієнти або кодування взаємодій між змінних для зворотного розповсюдження.

Без CNTK необхідно самому обирати процедуру оптимізації та розв’язувати похідні від функції витрат по відношенню до параметрів, які ми хочемо вивчити. Виходячи з логістичної регресії як імовірнісної моделі, ми можемо максимально збільшити ймовірність отримання даних. Це виявляється таким самим, як мінімізація функції перехресної ентропії, яка для логістичної регресії також відома як «функція логістичної втрати». Загалом, це не може бути зроблено аналітично, але можливо визначити аналітичні рішення для градієнтів, а потім використовувати градієнтне сходження, щоб сходити до правильних параметрів.

CNTK використовує загальний алгоритм стохастичного градієнтного спуску (SGD) для вивчення параметрів в обчислювальній мережі. Градієнти визначаються шляхом автоматичної диференціації.

Щоб налаштувати та навчити Комп'ютерну мережу, CNTK використовує файл конфігурації .cntk, який

1. описує мережу;

2. вказує команди, які ми хочемо виконати в мережі (тренування, тест, отримання вихідних значень і т. д.);

3. встановлює, як ми хочемо вивчити параметри мережі (SGD та її параметри);

4. як CNTK слід читати та записувати дані.

Для опису нашої обчислювальної мережі використовується мова опису мережі CNTK BrainScript. Потрібно визначити:

1. вхідні функції
2. позначення
3. параметри для навчання
4. обчислювальні операції
5. кореневі вузли (виходи)

Припустимо, що необхідно визначити обчислювальну мережу, яка виглядає так, як представлено на рисунку 2.9. Цю мережу можна зрозуміти як комбінацію трьох лінійних моделей, кожна з яких буде навчена відокремлювати один з трьох класів від двох інших. Потім, на виході, є шар *softmax*, який розподіляє лінійні моделі на за розподілом імовірності. Таким чином, для кожного входу мережа виводить три значення вірогідності, що в сумі дає 1. Наприклад, якщо даний екземпляр складається з класу (1), модель може виводити наступні ймовірності (95%, 3%, 2%), що в принципі означає, що ймовірність екземпляру, що відноситься до класу (1), становить 95% .

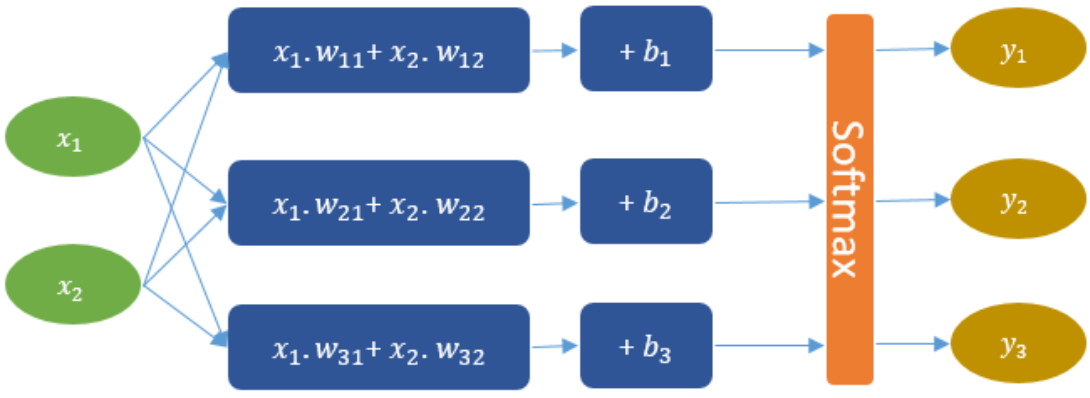


Рисунок 2.9 – Обчислювальна мережа [12]

Реалізація мережі, зображеної вище, за допомогою CNTK представлена на рисунку 2.10:

```
BrainScriptNetworkBuilder=[

    # sample and label dimensions
    SDim = $dimension$      # defined as 2 outside
    LDim = $labelDimension$ # defined as 3 outside

    features = Input {SDim}
    labels   = Input {LDim}

    # parameters to learn
    b = ParameterTensor {LDim}
    w = ParameterTensor {(LDim:SDim)}

    # operations
    z = w * features + b

    ce = CrossEntropyWithSoftmax (labels, z)
    errs = ErrorPrediction (labels, z)

    # root nodes
    featureNodes    = (features)
    labelNodes      = (labels)
    criterionNodes  = (ce)
    evaluationNodes = (errs)
    outputNodes     = (z)
]
```

Рисунок 2.10 – Реалізація нейронної мережі за допомогою CNTK [12]

CNTK також дозволяє працювати із CNN. CNN має більш складну структуру, ніж звичайні нейронні мережі. Кожен вхід тут є матрицею, а не вектором. Це відбувається тому, що CNN використовує локальні кореляції на зображенні. Таким чином, ми повинні зберегти цю інформацію. Шар згортки можна визначити як каскад шарів згортки та *max-pooling*. Реалізація у CNTK представлена на рисунку 2.11: [12]

```
ConvReLULayer (inp, outMap, inWCount, kW, kH, hStride, vStride) = [
    convW = Parameter (outMap, inWCount, init="uniform", initValueScale=wScale)
    convB = ImageParameter (1, 1, outMap, init="fixedValue", value=bValue,
imageLayout="$imageLayout$")
    conv = Convolution (convW, inp, kW, kH, outMap, hStride, vStride, zeroPadding=false,
imageLayout="$imageLayout$")
    act = RectifiedLinear (conv + convB)
].act
```

Рисунок 2.11 – Реалізація згорткового шару на CNTK [12]

2.2.4 PyTorch

PyTorch - це пакет python, який надає дві основні функції:

1. Тензорні обчислення (наприклад, numpy) з сильним прискоренням GPU
2. Глибокі нейронні мережі, побудовані на потоковій системі Autodiff

Зазвичай PyTorch використовують як заміна numpy, щоб використати можливості графічного процесора; і платформу для машинного навчання, яка є дуже гнучкою.

PyTorch надає тензори (рисунок 2.12), які можуть жити як на центральному процесорі, так і на графічному процесорі, і прискорити обчислення з великою кількістю даних.

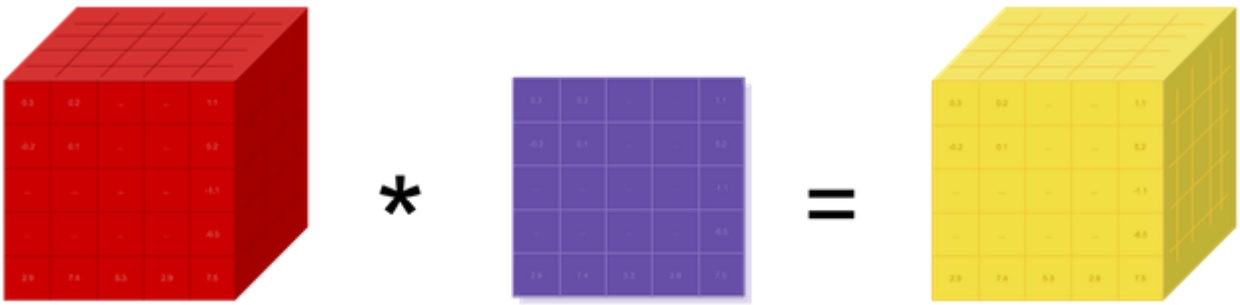


Рисунок 2.12 - Схематичне зображення тензору [13]

Одні з переваг PyTorch:

1. Немає необхідності писати код обгортки для PyTorch.
2. Легко налагоджувати та розуміти код
3. Має стільки типів шарів, як Torch (Unpool, CONV 1,2,3D, LSTM, Grus)
4. Має вбудовані фукнції втрат.
5. Може розглядатися як Numpy розширення для GPU
6. Дозволяє створювати мережі, структура яких залежить від самого обчислення (корисно для навчання з підсиленням)

PyTorch має унікальний спосіб створення нейронних мереж: використання та відтворення стрічки запису.

Більшість бібліотек, таких як TensorFlow, Theano, Caffe та CNTK, мають статичний вигляд. Потрібно побудувати нейронну мережу і повторно використовувати ту саму структуру знову і знову. Зміна способу поведінки мережі означає, що треба починати з нуля.

PyTorch використовує техніку, яка називається зворотньою автодиференціацією, яка дозволяє змінювати те, як працює мережа без додаткових зусиль. Ця техніка не є унікальною для PyTorch, але це одна з найшвидших його реалізацій на сьогодні. PyTorch дозволяє отримати найкращу швидкість і гнучкість для досліджень. Граф обчислень будується динамічно під час проходження мережі (рисунок 2.13).

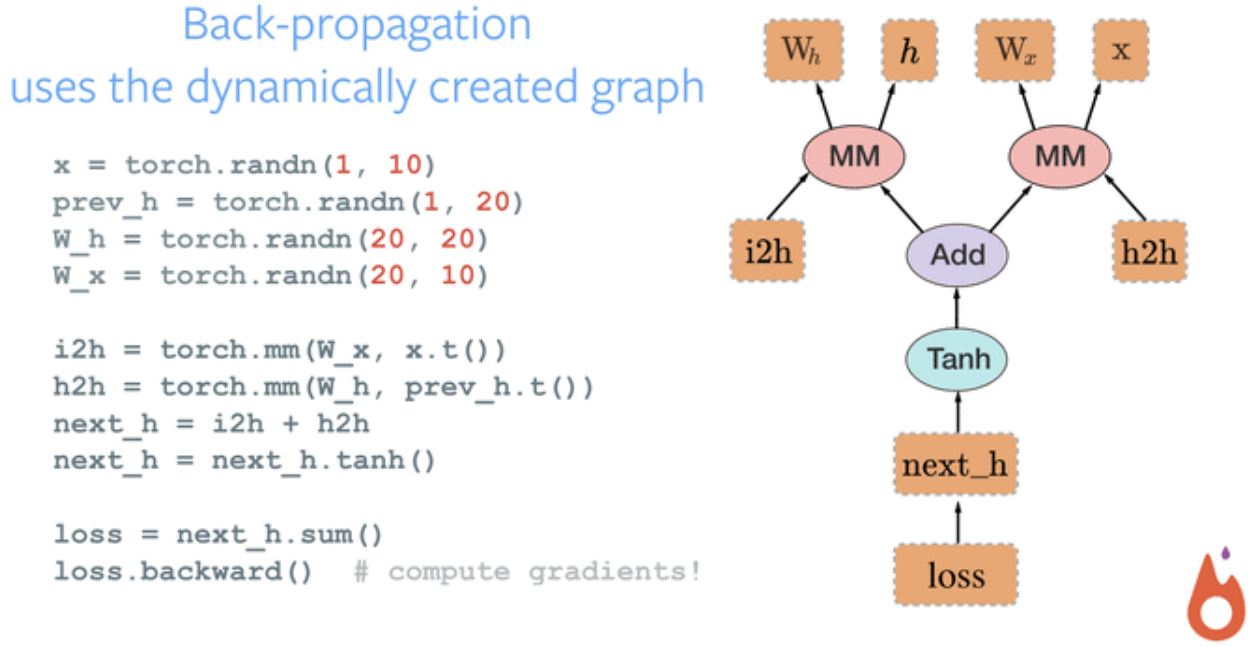


Рисунок 2.13 – Граф обчислень, побудований PyTorch [13]

PyTorch - це не обгортка Python над C++. Він побудований так, щоб бути глибоко інтегрованим в Python. Він може бути використаний це природно, як numpy / scipy / scikit-learn і т. д. Є можливість написати нові шари для нейронної мережі в Python, використовуючи існуючі бібліотеки та використовувати такі пакети, як Cython та Numba.

PyTorch розроблений, щоб бути інтуїтивно зрозумілим, лінійним та легким у використанні. Він використовує імперативний підхід, що означає, що коли виконується рядок коду, то він запускається одразу. Немає асинхронного погляду на світ. Якщо використовувати налагоджувач або отримути повідомлення про помилки та стеки виконання, то розуміння їх є прямим. Стек вказує на те, де саме код був визначений.

PyTorch має мінімальні накладні витрати. Він інтегрує прискорювальні бібліотеки, такі як Intel MKL і NVIDIA (CuDNN, NCCL), щоб максимально збільшити швидкість. PyTorch є досить швидким - чи використовуєте ви малі чи великі нейронні мережі.

Використання пам'яті в PyTorch є надзвичайно ефективним у порівнянні з Torch або деякими альтернативами. Було створено спеціальні розподільники пам'яті для GPU, щоб переконатися, що глибокі моделі навчання максимально ефективні для пам'яті. Це дає змогу навчати більші глибинні моделі навчання, ніж раніше.

Написання нових модулів для нейронних мереж або взаємодія з PyTorch's Tensor API було розроблено так, щоб бути прямим та з мінімальним абстракціями. Модель на PyTorch представлена на рисунку 2.14. [13]

```
def forward(self, x):
    #Computes the activation of the first convolution
    #Size changes from (3, 32, 32) to (18, 32, 32)
    x = F.relu(self.conv1(x))
    #Size changes from (18, 32, 32) to (18, 16, 16)
    x = self.pool(x)
    #Reshape data to input to the input layer of the neural net
    #Size changes from (18, 16, 16) to (1, 4608)
    #Recall that the -1 infers this dimension from the other given dimension
    x = x.view(-1, 18 * 16 *16)
    #Computes the activation of the first fully connected layer
    #Size changes from (1, 4608) to (1, 64)
    x = F.relu(self.fc1(x))
    #Computes the second fully connected layer (activation applied later)
    #Size changes from (1, 64) to (1, 10)
    x = self.fc2(x)
    return(x)
```

Рисунок 2.14 – Модель нейронної мережі на PyTorch [13]

2.2.5 Caffe

Caffe - бібліотека глибокого машинного навчання. Розроблений Berkeley AI Research (BAIR). Основними особливостями Caffe є:

Зрозуміла архітектура заохочує застосування та інновації. Моделі та оптимізація визначаються конфігурацією без жорсткого кодування. Є можливість переключатись між процесором і графічним процесором, встановивши єдиний прапорець для тренувань на машині графічного процесора, а потім розгорнути на кластерах чи мобільних пристрої.

Розширюваний код сприяє активному розвитку. У перший рік Caffe він був розкритий більш ніж 1000 розробниками і багато істотних змін застосовано. Завдяки цим авторам система підтримує найсучасніші технології в коді та моделях.

Швидкість робить Caffe ідеальним для дослідницьких експериментів та розгортання. Caffe може обробляти понад 60 M зображення на день за допомогою одного NVIDIA K40 GPU *. Це 1 мс / зображення для виводу та 4 мс / зображення для навчання.

Співтовариство: Caffe вже володіє академічними дослідницькими проектами, прототипами запуску та навіть великомасштабними промисловими програмами у сфері зору, мовлення та мультимедіа.

Основні складові Caffe

Глибокі мережі - це композиційні моделі, які природно представлені як сукупність взаємопов'язаних шарів, які працюють над частинами даних. Caffe визначає мережу шар за шароми у власній схемі моделей. Мережа визначає всю модель знизу вгору від вхідних даних до втрат. Оскільки дані та похідні потоку через мережу відбуваються в прямому і зворотньому напрямі, Caffe зберігає, передає та управляє інформацією як blob-ами: blob - це стандартний масив і уніфікований інтерфейс пам'яті для цієї системи. Наступний шар стає основою як для моделі, так і для обчислень. Мережа представляється як колекція і зв’язки між шарами. Деталі blob-ів описують, яким чином інформація зберігається та передається у шарах і

мережах. Рішення налаштовується окремо для відокремлення моделювання та оптимізації.

Blob - обгортка фактичних даних, що обробляються і передаються Caffe, а також під капотом забезпечують можливість синхронізації між процесором і графічним процесором. Математично, blob - це N-розмірний масив.

Caffe зберігає та передає дані, використовуючи blob-и. Blob-и забезпечують єдиний інтерфейс пам'яті, що містить дані; наприклад, наборів зображень, параметри моделі та похідні для оптимізації.

Blob-и приховують обчислювальні витрати змішаних операцій на CPU чи GPU, синхронізуючи їх із хоста процесора на пристрій GPU, коли це необхідно. Пам'ять на хості та пристрої виділяється на вимогу для ефективного використання пам'яті.

Звичайні розміри blob-ів для наборів даних зображення - це число N * канал K * висота H * ширина W.

Кількість/ N - розмір набору даних. Пакетна обробка забезпечує кращу пропускну спроможність для обробки повідомлень та пристроїв. Для тренування ImageNet набір 256 зображень N = 256.

Канал / K - це вимірність ознак, наприклад для RGB зображень K = 3.

Багато прикладів Caffe є чотиривимірними з осями для прикладних програм для зображень, але цілком правильно використовувати blob-и для програм, які не використовують зображення. Наприклад, якщо потрібні повнозв’язні мережі, як-от звичайний багатошаровий персептрон, можна використовувати 2D-blob (sahpe(N, D)) і викликати InnerProductLayer.

Розміри параметра blob залежать від типу та конфігурації шару. Для згорткового шару з 96 фільтрами розміром 11 x 11 та 3 входами розмір блоку становить 96 x 3 x 11 x 11. Для внутрішнього множення/ повнозв’язного шару з 1000 вихідними каналами та 1024 вхідними каналами параметр блоку становить 1000 x 1024.

Обчислення шару та зв'язків.

Шар є сутністю моделі та фундаментальної одиниці обчислень. Шари згорта.ть фільтри, pool, беруть внутрішні результати множення, застосовують нелінійності,

такі як ReLU та Sigmoid та інші елементні перетворення, нормалізують, завантажують дані та обчислюють втрати, такі як softmax.

Шар приймає вхід через нижні з'єднання і робить вихід через верхні з'єднання (рисунок 2.15).

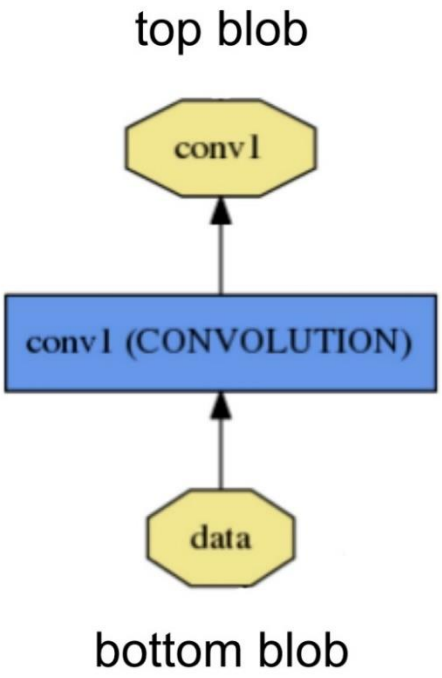


Рисунок 2.15 – Шар нейронної мережі у Caffe [14]

Кожен тип шару визначає три критичні обчислення: налаштування, пряме і зворотне проходження.

1. Налаштування: ініціалізація шару та його зв’язків проводиться один раз при ініціалізації моделі.

2. Пряме проходження: при заданому вході знизу обчислюється вихід і надсилається наверх.

3. Зворотне проходження: з урахуванням градієнта w.r.t. верхній вихід обчислює градієнт w.r.t. на вхід і відправляє вниз. Шар з параметрами обчислює градієнт w.r.t. до його параметрів і зберігає його всередині.

Шари мають дві основні функції для роботи в мережі в цілому: прямий прохід, який приймає входи і виробляє виходи, і зворотний прохід, який приймає градієнт

щодо виходу, і обчислює градієнти щодо параметрів і на входи, які в свою чергу поширюються назад до попередніх шарів.

Розробка користувацьких шарів вимагає мінімальних зусиль шляхом композиційної структури мережі та модульності коду. Визначте налаштування, пряме і зворотне проходження і цей шар готовий до включення в мережу.

Мережа спільно визначає функцію та її градієнт за композицією та автоматичною диференціацією. Композиція виводу кожного шару обчислює функцію для виконання заданого завдання, а композиція кожного шару зворотний обчислює градієнт з втрат для вивчення завдання. Моделі Caffe - це повноцінний движок для машинного навчання.

Мережа являє собою набір шарів, з'єднаних в обчислювальний граф – орієнтований ациклічний графік (DAG). Caffe виконує всю перевірку будь-якого DAG шарів, щоб забезпечити правильність прямого і зворотного проходження. Типова мережа починається з рівня даних, який завантажується з диска, і закінчується шаром втрат, який обчислює ціль для задачі, такої як класифікація або реконструкція.

Мережа визначається набором шарів та їх зв'язків на мові моделювання відкритого тексту. Простий логістичний регресійний класифікатор зображено на рисунках 2.16, 2.17:

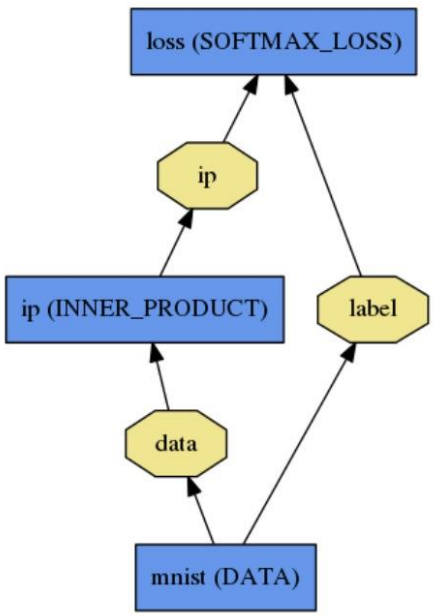


Рисунок 2.16 – Схема логістичного регресійного класифікатора [14]

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

Рисунок 2.17 – Реалізація логістичного регресійного класифікатора за допомогою Caffe [14]

Ініціалізація моделі обробляється Net :: Init (). Ініціалізація переважно полягає в двох аспектах: складанні загальної DAG, створюючи blob-и та шари і викликає функцію SetUp () шарів. Він виконує перевірку правильності загальної архітектури мережі.

Побудова мережі є незалежною від машини, де вона буде використовуватись, - blob-и та шари приховують деталі виконання з визначенням моделі. Після побудови мережа запускається на центральному або графічному процесорі, встановлюючи один перемикач, визначений у *Caffe :: mode()* і встановлений *Caffe :: set_mode ()*. Шари поставляються з відповідними процедурами процесора та графічного процесора, які виробляють однакові результати (аж до числових помилок і з тестами для його захисту). Перемикач CPU / GPU є незалежним від визначення моделі. Як для дослідження, так і для розгортання найкраще поділити модель та реалізацію.

Моделі визначаються в схемі буферів протоколу відкритого тексту (*prototxt*), тоді як навчені моделі серіалізуються за домогою бінарного протоколу буферу (binaryproto) у файлах .caffemodel.

Формат моделі визначається схемою protobuf у caffe.proto.

Caffe проводить комунікацію з Google Protocol Buffer для наступних сильних сторін: мінімальні розміри двійкових рядків під час серіалізації, ефективної серіалізації, текстового формату, який може читає людина, сумісного з бінарною версією, а також ефективні реалізації інтерфейсів на багатьох мовах, зокрема C ++ та Python. Це все сприяє гнучкості та розширюваності моделювання в Caffe. [14]

2.3 Висновки за розділом

Машинне навчання є дуже розповсюдженим на сьогодні, тому існує велика кількість бібліотек для розв’язку задач, що пов’язані з машинним навчанням. Головною відмінністю між бібліотеками є підхід, який вони використовують для виконання операцій: імперативний або символічний. Імперативний підхід є більш «натуральним», оскільки операції виконуються в той час, коли вони викликаються. Програми, які написані за допомогою такого підходу є більш гнучкими до змін. Символічний підхід полягає у відкладеному виконанні; при виконанні символічного коду спочатку складається граф обчислень (який можливо в подальшому оптимізувати), а вже потім цей граф заповнюється реальними даними, що використовуються у операціях. Перевагою такого підходу є те, що він швидший і більш економний за пам’яттю, оскільки може зберігати проміжні результати виконання (для однакових вхідних даних) і може оптимізувати граф обчислень.

У цьому розділі були розглянуті різні бібліотеки, що сьогодні використовуються розробниками, які працюють із задачами машинного навчання. Усі вони є різними за ступенями складності створення моделі, підходом виконання операцій. Ці відмінності мають впливати на час роботи бібліотеки під час виконання задачі розпізнавання.

3 ВИБІР КРИТЕРІЇВ ПОРІВНЯННЯ ТА ОЦІНКА БІБЛІОТЕК МАШИННОГО НАВЧАННЯ ЗА НИМИ

Спочатку визначимо задачу, на прикладі якої буде проводитись порівняння найпопулярніших фреймворків машинного навчання. Нехай, нам необхідно побудувати систему розпізнавання певних об’єктів на зображеннях. Оберемо такі об’єкти, що присутні у публічних датасетах, наприклад об’єкти з CIFAR-10 датасету [15].

Даний набір розділений на п'ять навчальних партій та одну тестову партію, кожна з яких складається з 10000 зображень. Тестовий пакет містить рівно 1000 випадкових зображень з кожного класу. Тренувальні партії містять залишкові зображення у випадковому порядку, але деякі тренувальні партії можуть містити більше зображень з одного класу, ніж інші. Між ними навчальні партії містять рівно 5000 зображень з кожного класу. Приклад зображень із цього набору представлено на рисунку 3.1.

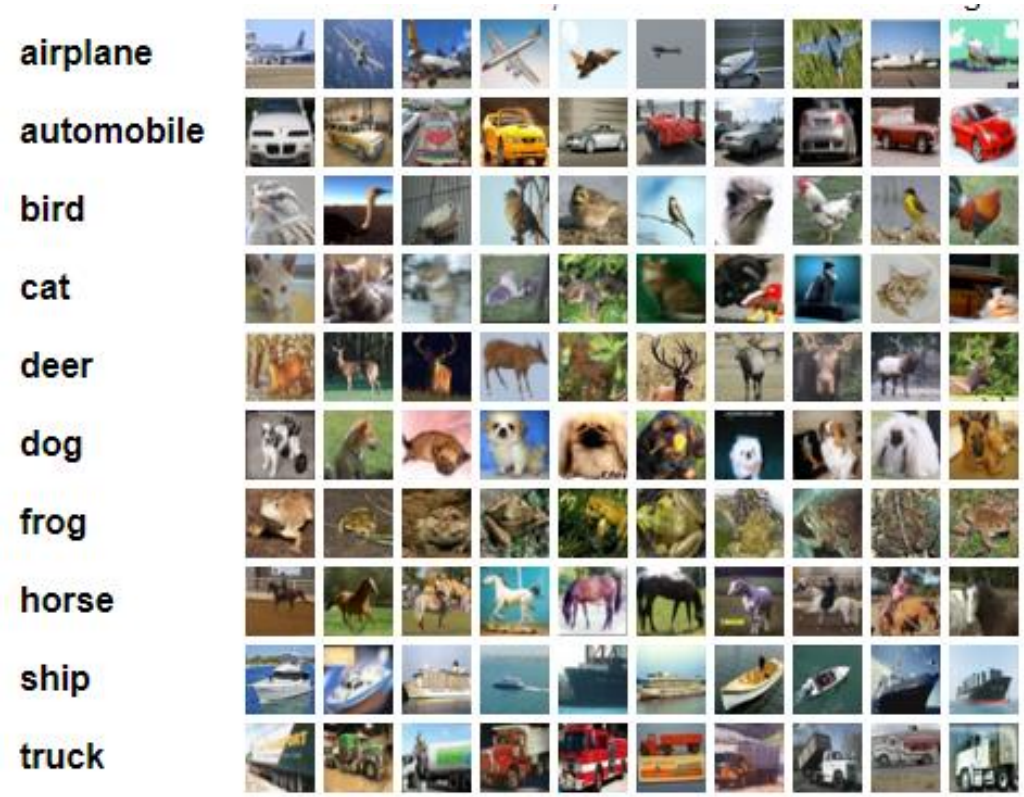


Рисунок 3.1 – Датасет CIFAR-10 [15]

3.1 Аналіз існуючих критеріїв порівняння

Очевидно, що в мережі присутня велика кількість порівнянь між бібліотеками машинного навчання для розв’язку задач розпізнавання. Критерії порівняння є різними, наприклад, порівнюють за такими загальними критеріями як: можливість використовувати графічний процесор для навчання, мова програмування, яку підтримує фреймворк, платформи, на яких підтримується фреймворк, за тим, хто розробив цей фреймворк, за кількістю доступних прикладів (рисунок 3.2). [16, 17]

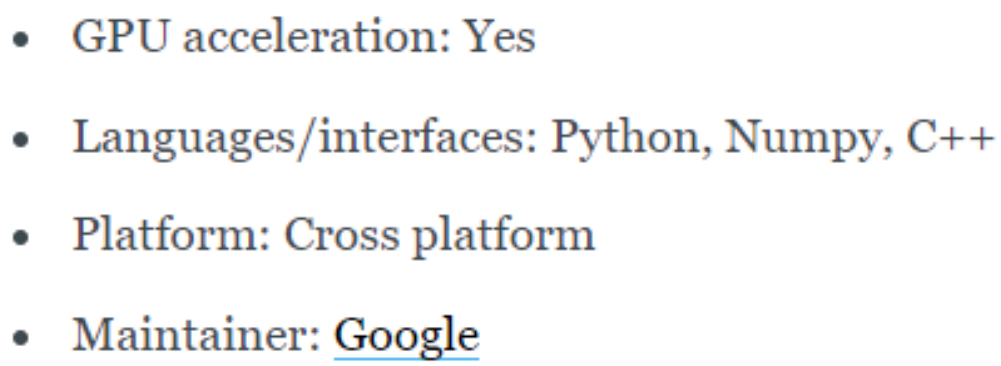


Рисунок 3.2 – Приклад порівняння бібліотек за загальними критеріями

Проте з таких загальних критеріїв неможливо зробити висновок, який саме фреймворк застосовувати, оскільки вони не відповідають на питання, чи складно створювати систему із застосуванням цього фреймворку, наскільки швидко навчатиметься система.

Більш цікавими критеріями є оцінки є можливість розподіленого виконання, способи оптимізації архітектури, візуалізація процесу навчання, підтримка, якою можна заручитись під час розробки (рисунок 3.3).

Framework	Distributed Execution	Architecture Optimisations	Visualisations	Community Support	Portability
TensorFlow	XX	XX	XX	XX	XX
PyTorch	XX	XX	XX	XX	XX
CNTK	XX	XX	X	-	XX
MXNet	X	XX	X	-	XX
Torch	-	XX	X	X	X
Caffe2	XX	XX	-	-	XX
Caffe	-	XX	X	X	X
Theano	-	XX	X	X	X

Рисунок 3.3 – Порівняння бібліотек за критеріями розподіленого виконання, оптимізації архітектури, візуалізації, підтримки [18]

Вже за цими критеріями можна робити перші висновки, який фреймворк застосовувати для розв’язку конкретної задачі. У даній роботі вище перераховані критерії були доповнені та оцінені.

Найбільш цікавим критерієм звичайно є швидкість роботи фреймворка, та швидкість навчання для досягання певної точності (зазвичай це 90 - 95%) (рисунок 3.4 і 3.5).

Для оцінки за цим критерієм проводять навчання на вже готовій нейронній мережі, для певного тренувального набору на одному й тому самому комп’ютері з використанням графічного процесора та без нього. Окрім загального часу виконання також заміряють час, який затрачено на розпізнавання кожного зображення окремо, та навіть вартість навчання у грошову еквіваленті (оренда серверів для навчання, затрачена електроенергія тощо).

DL Library	K80/CUDA 8/CuDNN 6	P100/CUDA 8/CuDNN 6
Caffe2	148	54
Chainer	162	69
CNTK	163	53
Gluon	152	62
Keras(CNTK)	194	76
Keras(TF)	241	76
Keras(Theano)	269	93
Tensorflow	173	57
Lasagne(Theano)	253	65
MXNet	145	51
PyTorch	169	51
Julia – Knet	159	*

Average Time(s) for 1000 Images: ResNet-50 – Feature Extraction

Рисунок 3.4 – Порівняння часу виділення ознак для ResNet-50 [19]

Objective: Time taken to train an image classification model to a test accuracy of 94% or greater on CIFAR10.

Rank	Time to 94% Accuracy	Model	Framework	Hardware
1 Apr 2018	0:02:54	Custom Wide Resnet *fast.ai + students team: Jeremy Howard, Andrew Shaw, Brett Koonce, Sylvain Gugger* source	fastai / pytorch	8 * V100 (AWS p3.16xlarge)
2 Apr 2018	0:05:41	Resnet18 + minor modifications *bkj* source	pytorch 0.3.1.post2	V100 (AWS p3.2xlarge)
3 Apr 2018	0:06:45	Custom Wide Resnet *fast.ai + students team: Jeremy Howard, Andrew Shaw, Brett Koonce, Sylvain Gugger* source	fastai / pytorch	Paperspace Volta (V100)
4 Apr 2018	0:35:37	KervResNet34 *Chen Wang* source	PyTorch 0.3.1	1 GPU (Nvidia GeForce GTX 1080 Ti)
5 Jan 2018	1:07:55	ResNet50 *DIUX* source	tensorflow 1.5, tensorpack 0.8.1	p3.2xlarge

Рисунок 3.5 – Порівняння часу тренування мережі для досягнення точності 94% [20]

Така оцінка вже краще дає розуміння, що краще обрати для розв’язку конкретної задачі розпізнавання. Проте усі ці порівняння не структуровані, проведені у різних умовах, що не дає можливості об’єктивно оцінити кожен з фреймворків. Тому було прийнято рішення провести заміри у однакових умовах і структурувати результати так, щоб стало можливим дійсно правильно порівняти найпопулярніші фреймворки між собою.

До того ж, не було знайдено жодної оцінки за виконанням базових операцій для фреймворків машинного навчання. Вони є також важливими, адже вони використовуються не тільки для розв’язку задачі розпізнавання, а й проведення великих за об’ємом обчислень, попередньої обробки даних (нормалізація, приведення до однакового виду). До того ж, при виборі активаційної функції для кожного шару нейронної мережі можна оцінити час, за який вона опрацьовує різну кількість даних, оскільки час розпізнавання очевидно залежить також і від часу роботи активаційної функції.

3.2 Загальні критерії порівняння

Для оцінки було виділено такі критерії порівняння:

Спосіб опису архітектури нейронної мережі. У цьому критерії важливо, щоб людина, яка не знайома з даним фреймворком, проте знайома із галуззю машинного навчання, могла легко розуміти, що за архітектура використовується у даній системі: які шари, як вони зв’язані, які активаційні функції використовуються на кожному шарі.

Можливість переналаштування архітектури мережі. Цей критерій є важливим, оскільки швидке переналаштування мережі дозволяє легко проводити експерименти з різними варіантами нейронних мереж, особливо тоді, коли спочатку точно не зрозуміло, яку кількість шарів необхідно використовувати, скільки нейронів має бути в кожному шарі, які активаційні функції використовувати і так далі.

Можливість навчання на графічному процесорі. Цей критерій полягає у тому, чи підтримує фреймворк платформу CUDA [21], що значно пришвидшує час навчання та дозволяє виконувати операції паралельно.

Доступність прикладів. Кожна бібліотека має документацію, за допомогою якої розробник розуміє, як йому використовувати ту чи іншу бібліотеку. Проте найкращим способом старту роботи з бібліотекою – подивитись, як вона працює на конкретних прикладах, спробувати самому в них щось змінити, наприклад так, щоб реалізація була найбільш вдалою для розв'язку конкретної задачі. Очевидно, що робота вже з готовими прикладами буде швидшою і простішою, ніж написання коду з нуля.

Підтримка попередньо натренованих моделей: відомо, що в деяких випадках використання попередньо натренованих моделей дозволяють покращити результати навчання, ніж навчання мережі з нуля. Тому важливо, щоб бібіліотека мала змогу працювати з такими моделями, щоб розробник мав змогу провести експеримент, як його система буде працювати краще: з використанням попередньо натренованих моделей, чи без них.

Підтримка різних ОС. Цей критерій є найменш суттєвим, проте іноді умови бувають такими, що неможливо використовувати, наприклад, операційну систему Linux, оскільки у користувача вже в наявності є сервери, що використовують Windows.

Важливість кожного критерія була оцінена у таблиці 3.1.

Таблиця 3.1 – Оцінка важливості кожного критерію

Критерій	Оцінка (0-5)
Архітектура	5
Переналаштування	5
Підтримка попередньо натренованих моделей	3
Наявність прикладів	3
Тренування	4
ОС	2
Паралельні обчислення	4

3.3 Критерій часу виконання базових операцій

Для порівняння було обрано ті базові операції, які найчастіше використовуються під час тренування нейронної мережі, а саме:

Сума і множення матриць. Використовуються під час перемноження входів (inputs) та ваг (weigths), суми ваг та зміщення (bias), операції згортки (convolution) і т.д.. Матриці можуть бути довільної розмірності.

Транспонування матриць. Оскільки в нейронній мережі необхідно обробляти ваги та входи довільної розмірності, то ці матриці можуть не задовольняти правилам множення. Саме тому матриці доводиться часто транспонувати.

Серед активаційних функцій було обрано ReLU (recti-fied linear unit), яка представлена у та Sigmoid.

Для кожної епохи навчання розраховують функцію втрат (loss function) – функцію, яку мінімізують алгоритми машинного навчання. За допомогою функції втрат оцінюють якість тренування мережі. Одними з реалізацій функції втрат є reduce sum, що обраховує суму усіх елементів матриці за заданими або усіма осями, або reduce mean – визначення середнього значення послідовності.

3.4 Критерій часу, за який система набуде необхідної точності під час розв’язку задачі розпізнавання

Найважливішим критерієм для порівняння обраних бібліотек звичайно є те, як довго кожен з них буде проводити тренування мережі, тому було окреслено умови, за яких буде проводитись навчання.

Мову програмування обрано Python, оскільки для неї фреймворки представлені найширше.

Зображення з тренувального набору було оброблено у такі способи:

1. Кожне зображення було обрізано до розміру 24x24 пікселі відносно центра під час оцінки точності роботи нейронної мережі та обрізано довільно під час тренування.

2. Кожен зображення було приблизно вирівняне за гістограмою, та перетворено у чорно-біле, щоб зробити нейронну мережу не сприйнятливою до коливань яскравості і кольору.

Також деяка кількість зображень (15%) була випадковим чином віддзеркалена, змінена яскравість та контрастність зображення з метою покращення точності роботи мережі.

Далі було обрано структуру мережі виходячи з поставленої задачі. Оскільки це задача розпізнавання зображень, то для своєї системи оберемо convolutional neural network (згорткову нейронну мережу), що найкраще справляється з таким типом задач через особливості своєї архітектури. Схема архітектури представлена на рисунку 3.6.

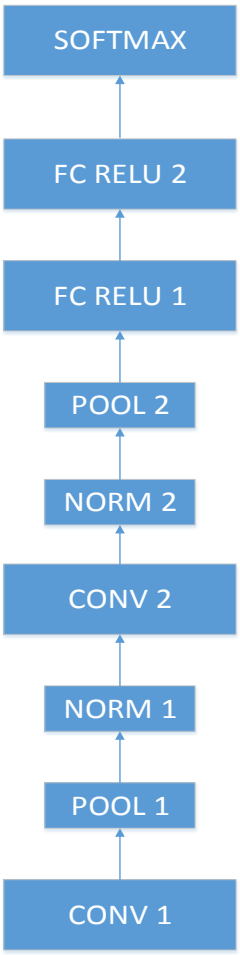


Рисунок 3.6 – Схема обраної архітектури нейронної мережі

Опис кожного шару представлено у таблиці 3.2.

Таблиця 3.2 – Опис шарів, що використовуються у нейронній мережі.

CONV 1, CONV 2	Згортка (convolution) над вхідними даними з використанням вказаного фільтру, і над отриманим результатом використовується нелінійність ReLU
POOL 1, POOL 2	Max pooling - обчислює середнє значення кожної підматриці, яка обмежена границями певного розміру, які здвигаються на заданий крок
NORM 1, NORM 2	Local response normalization - вхідний 4-D тензор використовується як 3-D масив одиничних векторів, і кожний вектор нормалізується окремо (тобто кожен компонент вектора ділиться на квадрат суми усіх компонентів)
FC 1, FC 2	Повнозв'язний шар, з активаційною функцією ReLU
Softmax	Шар, що відносить вхідні дані до певного класу, застосовує класифікатор Softmax

Для навчання нейронної мережі було обрано метод логістичної регресії, або softmax-регресії. Softmax-регресія застосовує softmax нелінійність для виходів нейронної мережі і розраховує перехресну ентропію між нормалізованими прогнозованими виходами і індексами визначених класів. Як функція втрат для даної моделі було використано суму втрат перехресної ентропії (формула 3.1).

$$H(X,Y) = \sum p(x) * \log_2(q(y)) \tag{3.1}$$

де p(x) – ймовірність появи значення x змінної X,

q(y) – ймовірність появи значення y змінної Y.

3.5 Результати порівняння за загальними критеріями

PyTorch. Це фреймворк, який дозволяє проводити обчислення швидко на графічному процесорі та будувати нейронні мережі. Переваги: підтримка попередньо натренованих моделей, швидка зміна архітектури нейронної мережі завдяки техніці reverse-mode auto-differentiation (автоматичне диференціювання з рухом назад), підтримка візуалізації побудованої архітектури, опис моделі за допомогою методів, що мають назви шарів згорткової нейронної мережі, підтримка візуалізації побудованої архітектури, опис моделі за допомогою методів, що мають назви шарів згорткової нейронної мережі (conv, relu, max_pool), що пришвидшує початок роботи з цим фремворком, підтримка CUDA. До недоліків можна віднести те, що код для тренування необхідно писати самому, що сповільнює розробку.

TensorFlow. Фреймворк від Google, який замінив Theano, став простішим і швидшим на відміну від попередника, є найпопулярнішим серед розробників. Переваги: архітектура мережі описується у термінах, що використовуються у CNN, має детальну візуалізацію TensorBoard, що будує граф під час навчання, готова реалізація тренування, використовує auto-differentiation для переналаштування архітектури, підтримує паралельні обчислення, підтримка CUDA. Недоліки: значно повільніший за інші фреймворки (наприклад, MxNet), невелика кількість попередньо натренованих моделей, складний для початківця.

Caffe. Фреймворк, який спеціалізується саме на розпізнаванні зображень. Переваги: підтримка CNN, велика кількість попередньо натренованих моделей, опис моделей відбувається за допомогою JSON, що дозволяє краще розуміти вкладеність та структуру шарів без візуалізації, або за допомогою вбудованих методів, присутня візуалізація, типи шарів описуються у відомих термінах для CNN, тренування проводиться без написання додаткового коду, лише за допомогою файлів налаштувань, підтримка CUDA. Недоліки: важко розширювати для виконання інших задач розпізнавання повільно працює з великими мережами, які не були попередньо натреновані, відсутність швидкого переналаштування, навчання проводитиметься спочатку, майже відсутня підтримка.

MXNet. Фреймворк для машинного навчання від Microsoft і Amazon. Переваги: набагато швидший за інші фреймворки (швидше навчання, менше використання оперативної пам’яті), підтримує попередньо натреновані моделі (у прикладах пристуній готовий код для fine-tuning), опис моделі проводиться у термінах CNN, присутня візуалізація, можливе швидке переналаштування завдяки auto-differentiation, готовий код для тренування, підтримка CUDA. Недоліки: необхідно встановлювати окремі версії фреймоврка для GPU та CPU, невелика кількість документації.

Основні критерії порівняння винесені в таблицю 3.3. Варто додати, що усі фреймворки підтримують навчання на GPU за допомогою CUDA, та мають візуалізацію побудованої мережі. В результаті порівняння кожного фреймворку за обраними критеріями, їх було оцінено по калі від 0 до 5 (від найгіршого результату до найкращого). Результат цієї оцінки представлено в таблиці 3.4.

Таблиця 3.3 - Порівняння бібліотек за обраними критеріями

Бібліотека	Архітектура	Переналаштування	Підтримка попередньо натренованих моделей	Наявність прикладів	Трену-вання	ОС	Парале-льні обчис-лення
PyTorch	Методи, що мають назву шарів CNN	Auto-differentiation	Присутня велика кількість моделей, легке тренування	Сайт фреймворку, репозиторії на github	Треба писати самому	Windows, Linux	+
TensorFlow	Методи, що мають назву шарів CNN	Auto-differentiation	Не так розповсюджено	Сайт фреймворку, репозиторії на github, статті	Готові методи	Windows, Linux	+
CAFFE	Методи, що мають назву шарів CNN; JSON структура	Модель має бути перенатренована	Фреймворк націлений саме на fine-tuning	Репозиторії на github	Не потрібен код, тільки налаш-тування	Linux, формальна підтримка Windows	-
MXNet	Методи, що мають назву шарів CNN	Auto-differentiation	Присутня велика кількість моделей, легке тренування	Сайт фреймворку, репозиторії на github	Готові методи	Windows, Linux	+

Таблиця 3.4 - Оцінка кожного фреймворку за обраними критеріями

Бібліотека	Архітек-тура	Перенала-штування	Підтримка попередньо натренованих моделей	Наявність прикладів	Трену-вання	ОС	Пара-лельні обчи-слення	**Сума**
PyTorch	4.5	5	4	4	3	5	5	30.5
TensorFlow	4.5	5	3	5	5	5	5	32.5
CAFFE	5	0	5	3.5	5	4	0	22.5
MXNet	4.5	5	4.5	4	5	5	5	33

3.6 Результати за часом виконання базових операцій

Кожен з фреймворків було порівняно за часом виконання операцій. Конкретні результати тестів представлено після опису кожного з них. Заміри проводились на CPU Intel Core i7-3517U 2.4 GHz, 8 GB RAM, GPU NVidia GeForce 1060, OS Windows 10. Тестові скрипти були на мові програмування Python 3.6.4.

Tensorflow. Це символічний фреймворк з відкритим кодом для проведення обчислень з використанням графа обчислень та машинного навчання від Google Brain Team. Використовує виключно символічний підхід. На сьогодні є найпопулярнішим фреймворком, що використовується на багатьох проектах, хоча і створювати системи на ньому доволі важко через складний API.

Швидкість: із зростанням розмірності задачі, Tensorflow виконував операції все повільніше і повільніше, багато часу уходило на компіляцію (ініціалізацію сесії). Проте і на малій розмірності Tensorflow показав найгірші результати. Про повільне виконання зауважують різні ентузіасти, що порівнювали швидкість тренування для Tensorflow [22].

Пам’ять: під час виконання операцій було помітно, що пам’ять дещо зростала (300 MB – 500 MB – 800 MB – 1100 MB), а потім знову падала до початкового значення у 300 MB. Це означає, що Tensorflow нормально звільняє пам’ять під час виконання операцій.

Лістинг для тесту Tensorflow представлено на рисунку 3.7.

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for j in range(1000):
        start_time = time.time()
        sess.run(operation) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рисунок 3.7 - Лістинг для тесту Tensorflow

MXNet. Це фреймворк від Apache, який підтримує і символічний, і імперативний підходи. Гнучкий та швидкий фреймворк, який підтримується Microsoft і Amazon. Набирає популярність серед спеціалістів, що працюють у сфері машинного навчання.

Швидкість: із збільшенням розмірності задачі час виконання операцій на MXNet майже не змінювався, був відносно малим. У порівнянні між символічним та імперативним підходами символічний підхід показав результати в середньому у 4 рази. Показав найкращу швидкість виконання серед усіх.

Пам’ять: що для імперативного, що для символічного підходів пам’ять зростала не сильно (300 MB – 450 MB) та швидко поверталась до початкового значення, тобто MXNet також нормально звільняє пам’ять.

Лістинг для тесту MXNet представлено на рисунках 3.8 і 3.9.

```
def perform_operation(shape, mul):
    time_sum = 0
    a = mx.symbol.Variable('A')
    e = operation(a) # assign operation
    for j in range(1000):
        a_data = mx.nd.uniform(low=-deviation,
                               high=deviation, shape=[shape] * mul)
        executor = e.bind(mx.cpu(), {'A':a_data})
        start_time = time.time()
        executor.forward() # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рисунок 3.8 - Лістинг для тесту MXNet (символічний підхід)

```
def operation(shape, mul):
    time_sum = 0
    for j in range(1000):
        a = mx.nd.uniform(low=-deviation, high=deviation,
                          shape=[shape] * mul)
        start_time = time.time()
        mx.operation(a) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рисунок 3.9 - Лістинг для тесту MXNet (імперативний підхід)

PyTorch. Символічний фреймворк, Python версія фреймворку, був розроблений Facebook. Налаштований для проведення обчислень на GPU.

Швидкість: перевірявся тільки імперативний підхід, тому швидкість виконання зростала із збільшенням розмірності задачі, швидкість сильно зростала для множення матриць, обчислення Sigmoid та Reduce Sum. Транспонування, сума матриць і виконання функції ReLU були швидкими незалежно від розмірності задачі.

Пам’ять: через імперативний підхід із збільшенням задачі до розмірності чотиривимірних матриць відбувся витік пам’яті, що унеможливило перевірку на великих даних. Для малої розмірності задачі пам’ять трималась на рівні 400-500 MB. Проте про витік пам’яті є свідчення і на інших операціях, в тому числі і на GPU [23].

Лістинг для тесту PyTorch представлено на рисунку 3.10.

```
def perform_operation(shape, mul):
    time_sum = 0
    a = torch.randn([shape]*mul, dtype=torch.double)
    for j in range(1000):
        start_time = time.time()
        torch.operation(a) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рисунок 3.10 - Лістинг для тесту PyTorch

CNTK. Microsoft Cognitive Toolkit (або CNTK) фреймворк для машинного навчання від Microsoft, має покращені алгоритми для обробки великих датасетів. Не для всіх опрецій був представлений API, тому сума і транспонування матриць не представлена для цього фреймворку.

Швидкість: був використаний імперативний підхід, тому швидкість зростала із збільшенням розмірності задачі, проте в основному швидше, ніж PyTorch і набагато швидший за Tensorflow.

Пам’ять: через імперативний підхід пам’ять із збільшенням задачі не вивільнялась, тому це унеможливило провести заміри для чотиривимірних матриць через витік пам’яті. При виконання менших задач пам’ять трималась на рівні 600 MB – 800 MB.

Лістинг для тесту CNTK представлено на рисунку 3.11.

```
def perform_operation(shape, mul):
    time_sum = 0
    for j in range(1000):
        a = np.random.sample(size = [shape]*mul)
        start_time = time.time()
        cntk.operation(a) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рисунок 3.11 - Лістинг для тесту CNTK

Результати замірів**.** Тестування проводилось для матриць різних розмірностей: 10x10, 10x10x10, 10x10x10x10, 100x100, 100x100x100, 100x100x100x100, 1000x1000. Умовні позначення на кожному з графіків: «Add matrices (…)» - сума двох матриць, «Mul 2 matrcies (…)» - множення двох матриць, «Mean» - reduce mean - пошук середнього значення за усіма осями, «Relu» - обчислення значення функції ReLU, «Sigmoid» - обчислення значення функції Sigmoid, «Reduce sum» - виконання операції

reduce sum, «Transpose» - транспонування матриці. По вертикальній осі вказано час у секундах.

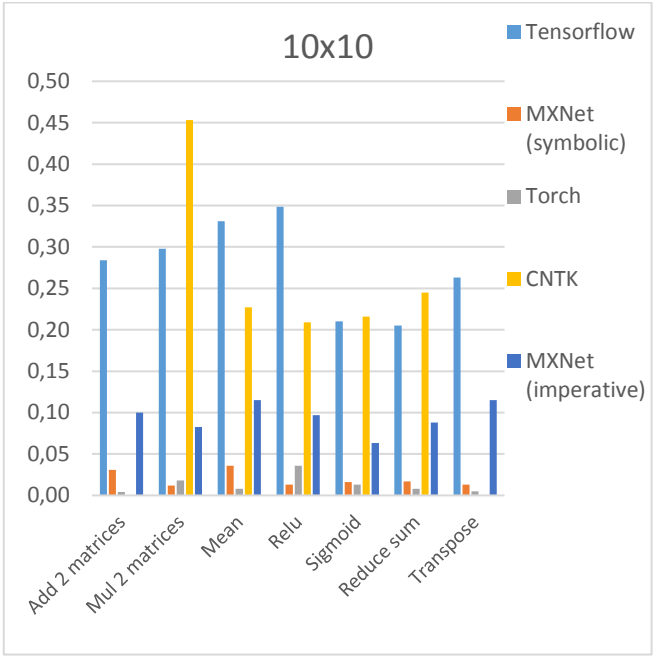


Рисунок 3.12 - Тест для матриці 10x10

З рисунка 3.12 видно, що вже на малих розмірностях MXNet працює швидше за інші фреймворки, проте Torch для операцій Reduce Sum, Mean та суми двох матриць і транспонування був швидшим, майже всюди найгірші результати показує Tensorflow, найгірше множення у CNTK.

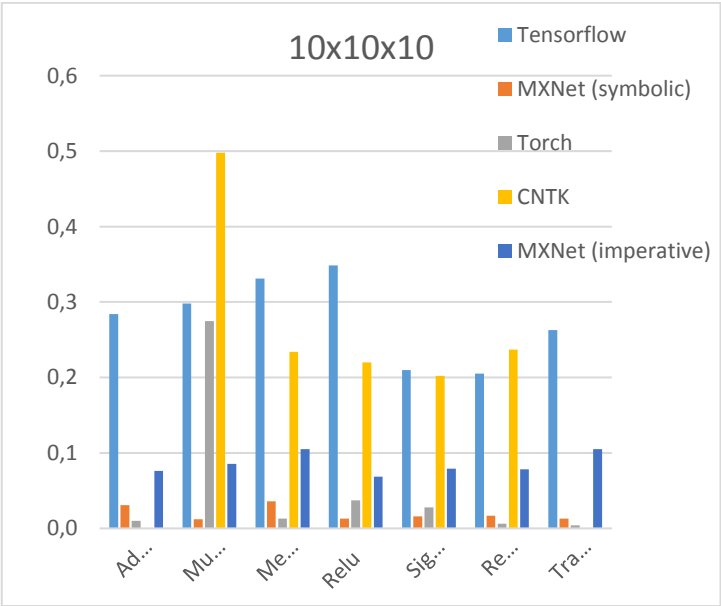


Рисунок 3.13 - Тест для матриці 10x10x10

З рисунка 3.13 видно, що MXNet залишається майже всюди найшвидшим, за винятками для PyTorch, що були описані вище. В основному, результат схожий на попередній.

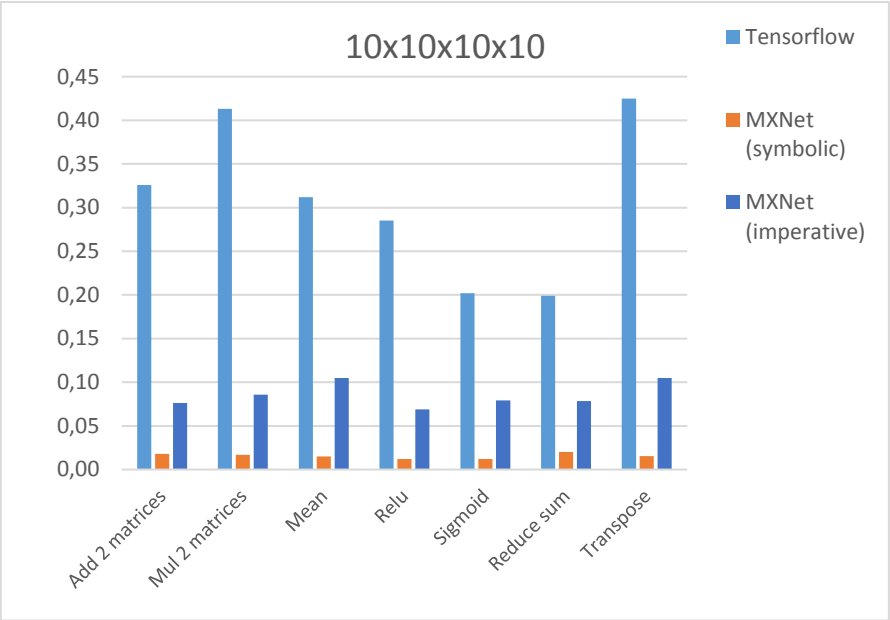


Рисунок 3.14 - Тест для матриці 10x10x10x10

З рисунка 3.14 видно, що результати для трьох представлених тестів схожі на попередній, проте для CNTK і PyTorch вже на такій розмірності відбувся витік пам’яті.

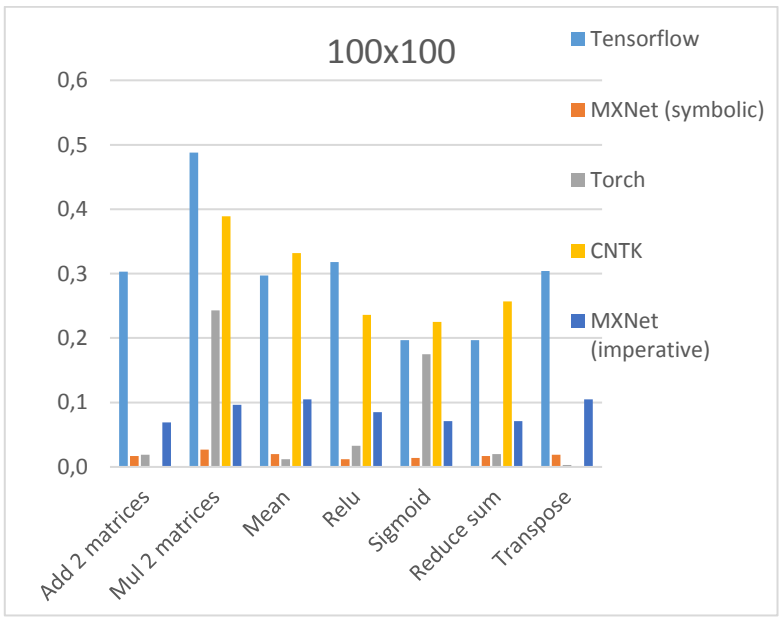


Рисунок 3.15 - Тест для матриці 100x100

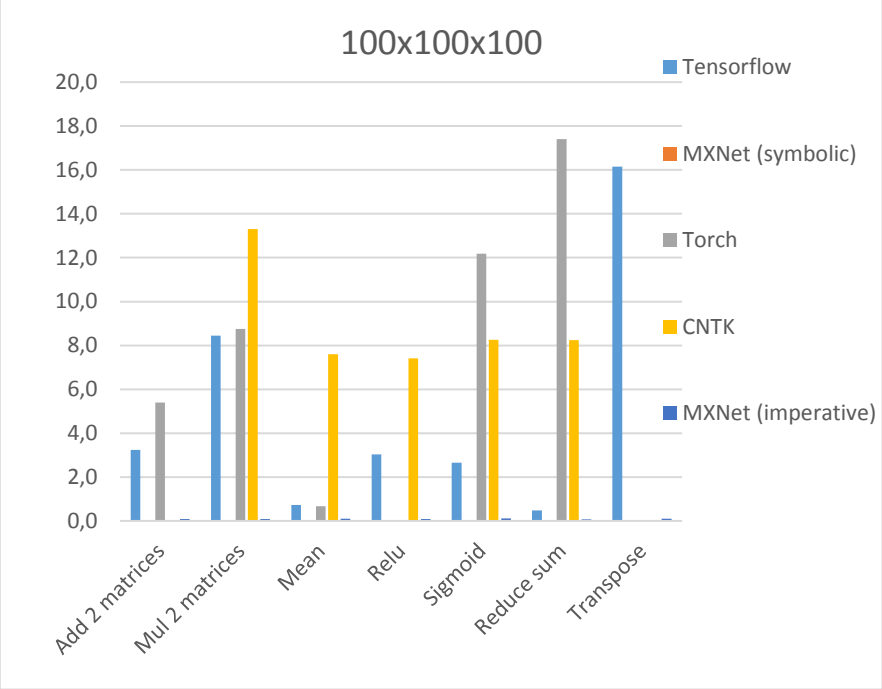


Рисунок 3.16 - Тест для матриці 100x100x100

З рисунків 3.15 і 3.16 видно, що MXNet на цих розмірностях вже набагато швидший за інші фреймворки, окрім операції транспонування, тут PyTorch кращий. Tensorflow поступово сповільнюється.

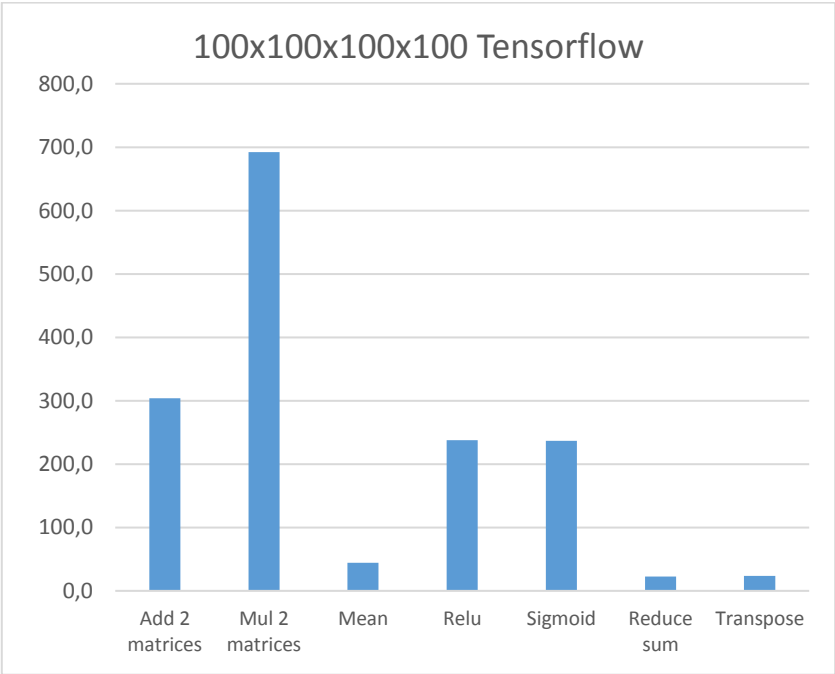


Рисунок 3.17 - Тест для матриці 100x100x100x100 для TensorFlow

З рисунку 3.17 видно, що Tensorflow вже дуже сильно сповільнюється, у порівнянні з MXNet. Для CNTK і PyTorch на цій розмірності стався витік пам’яті.

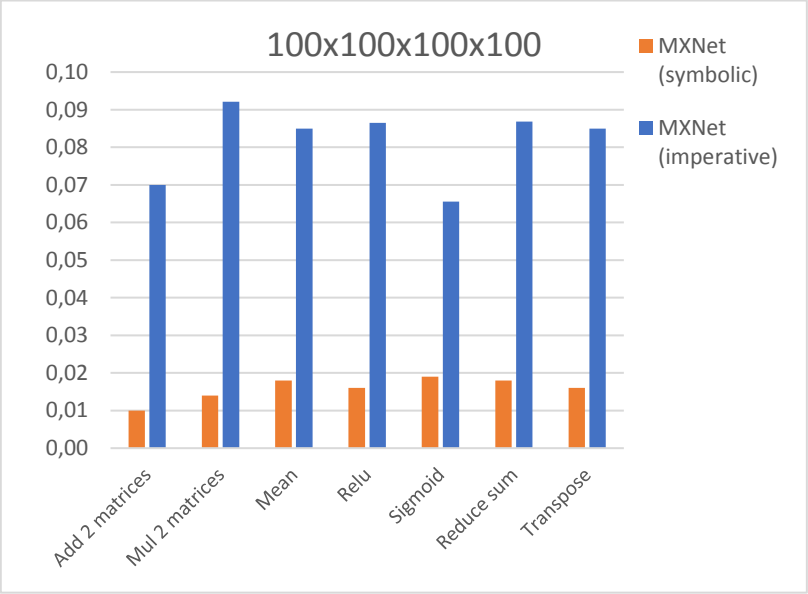
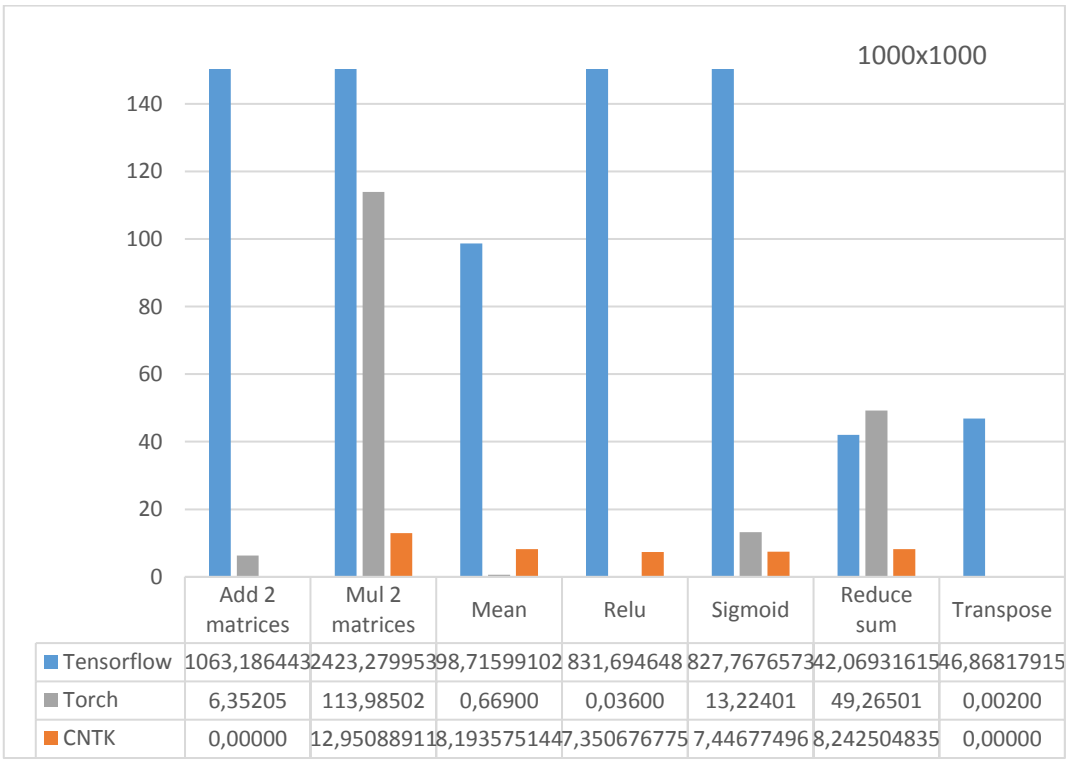


Рисунок 3.18 - Тест для матриці 100x100x100x100 для MXNet

З рисунку 3.18 видно, що MXNet стабільно швидко виконує операції для більшої розмірності, при чому символічний підхід набагато швидший за імперативний.



	Add 2 matrices	Mul 2 matrices	Mean	Relu	Sigmoid	Reduce sum	Transpose
Tensorflow	1063,186443	2423,279953	98,71599102	831,694648	827,7676573	42,06931615	46,86817915
Torch	6,35205	113,98502	0,66900	0,03600	13,22401	49,26501	0,00200
CNTK	0,00000	12,95088911	8,193575144	7,350676775	7,44677496	8,242504835	0,00000

Рисунок 3.19 - Тест для матриці 1000x1000

З рисунку 3.20 видно, що Tensorflow знову показує найгірші результати, PyTorch також став повільнішим. Для MXNet результати схожі на результати з рисунку 3.16.

За результатами досліджень було проведено оцінку фреймворків за часом виконання базових операцій, де 1 – найгірша оцінка, а 5 – найкраща, 0 – не проводився тест (таблиця 3.5). Умовні позначення у таблиці 1: «A+B» - сума матриць, «A dot B» - множення матриць, «A.T» - транспонування матриць.

Таблиця 3.5 - Оцінка фреймворків за часом виконання базових операцій

	A + B	A dot B	Reduce mean	Reduce sum	ReLU	Sigm	A.T	Σ
Tensorflow	1	1	3	3	2	2	2	14
MXNet	5	5	5	5	5	5	4	34
PyTorch	2	2	3	2	3	2	5	19
CNTK	0	2	2	3	2	3	0	12

3.7 Результати за часом навчання нейронної мережі відповідно до поставленої задачі

Для кожного фреймворку було побудовано мережу, схема якої була вказана у попередньому розділі. Оцінювався час, за який модель досягала точності у 90 %. Для кожного фреймворку представлена побудована модель.

PyTorch. Побудована модель представлена на рисунку 3.20.

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

Рисунок 3.20 – Модель нейронної мережі, реалізованої на PyTorch

Час, за який було досягнуто точність 90%: GPU 151 хв; CPU 482 хв.

TensorFlow. Побудована модель представлена на рисунках 3.21 і 3.22

```
def model(images):

 #conv1
          with tf.variable_scope('conv1') as scope:
            kernel = _variable_with_weight_decay('weights',
                                                 shape=[5, 5, 3, 64],
                                                 stddev=5e-2,
                                                 wd=None)
            conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
            biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.0))
            pre_activation = tf.nn.bias_add(conv, biases)
            conv1 = tf.nn.relu(pre_activation, name=scope.name)
            _activation_summary(conv1)

          # pool1
          pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                                 padding='SAME', name='pool1')
          # norm1
          norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                            name='norm1')
```

Рисунок 3.21 – Модель нейронної мережі, реалізованої на TensorFlow

```
# conv2
with tf.variable_scope('conv2') as scope:
  kernel = _variable_with_weight_decay('weights',
                                       shape=[5, 5, 64, 64],
                                       stddev=5e-2,
                                       wd=None)
  conv = tf.nn.conv2d(norm1, kernel, [1, 1, 1, 1], padding='SAME')
  biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.1))
  pre_activation = tf.nn.bias_add(conv, biases)
  conv2 = tf.nn.relu(pre_activation, name=scope.name)
# norm2
norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm2')
# pool2
pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1],
                       strides=[1, 2, 2, 1], padding='SAME', name='pool2')
# fc1
with tf.variable_scope('fc1') as scope:

  reshape = tf.reshape(pool2, [images.get_shape().as_list()[0], -1])
  dim = reshape.get_shape()[1].value
  weights = _variable_with_weight_decay('weights', shape=[dim, 384],
                                        stddev=0.04, wd=0.004)
  biases = _variable_on_cpu('biases', [384], tf.constant_initializer(0.1))
  Fc1 = tf.nn.relu(tf.matmul(reshape, weights) + biases, name=scope.name)
  _activation_summary(local3)
# fc2
with tf.variable_scope('fc2') as scope:
  weights = _variable_with_weight_decay('weights', shape=[384, 192],
                                        stddev=0.04, wd=0.004)
  biases = _variable_on_cpu('biases', [192], tf.constant_initializer(0.1))
  Fc2 = tf.nn.relu(tf.matmul(local3, weights) + biases, name=scope.name)
  _activation_summary(local4)


# linear softmax layer(WX + b),
with tf.variable_scope('softmax_linear') as scope:
  weights = _variable_with_weight_decay('weights', [192, NUM_CLASSES],
                                        stddev=1/192.0, wd=None)
  biases = _variable_on_cpu('biases', [NUM_CLASSES],
                            tf.constant_initializer(0.0))
  softmax_linear = tf.add(tf.matmul(fc2, weights), biases, name=scope.name)
  _activation_summary(softmax_linear)


return softmax_linear
```

Рисунок 3.22 – Продовження моделі нейронної мережі, реалізованої на TensorFlow

Час, за який було досягнуто точність 90%: GPU 175 хв; CPU 497 хв.

MXNet. Побудована модель представлена на рисунку 3.23.

```
def ConvFactory(data, num_filter, kernel, stride=(1,1), pad=(0, 0), act_type="rel
u"):
    conv = mx.symbol.Convolution(data=data, workspace=256,
                                 num_filter=num_filter, kernel=kernel, stride=str
ide, pad=pad)
    bn = mx.symbol.BatchNorm(data=conv)
    act = mx.symbol.Activation(data = bn, act_type=act_type)
    return act
def DownsampleFactory(data, ch_3x3):
    # conv 3x3
    conv = ConvFactory(data=data, kernel=(3, 3), stride=(2, 2), num_filter=ch_3x3
, pad=(1, 1))
    # pool
    pool = mx.symbol.Pooling(data=data, kernel=(3, 3), stride=(2, 2), pad=(1,1),
pool_type='max')
    # concat
    concat = mx.symbol.Concat(*[conv, pool])
    return concat

def SimpleFactory(data, ch_1x1, ch_3x3):
    # 1x1
    conv1x1 = ConvFactory(data=data, kernel=(1, 1), pad=(0, 0), num_filter=ch_1x1
)
    # 3x3
    conv3x3 = ConvFactory(data=data, kernel=(3, 3), pad=(1, 1), num_filter=ch_3x3
)
    #concat
    concat = mx.symbol.Concat(*[conv1x1, conv3x3])
    return concat

data = mx.symbol.Variable(name="data")
conv1 = ConvFactory(data=data, kernel=(3,3), pad=(1,1), num_filter=96, act_type="
relu")
in3a = SimpleFactory(conv1, 32, 32)
in3b = SimpleFactory(in3a, 32, 48)
in3c = DownsampleFactory(in3b, 80)
in4a = SimpleFactory(in3c, 112, 48)
in4b = SimpleFactory(in4a, 96, 64)
in4c = SimpleFactory(in4b, 80, 80)
in4d = SimpleFactory(in4c, 48, 96)
in4e = DownsampleFactory(in4d, 96)
in5a = SimpleFactory(in4e, 176, 160)
in5b = SimpleFactory(in5a, 176, 160)
pool = mx.symbol.Pooling(data=in5b, pool_type="avg", kernel=(7,7), name="global_a
vg")
flatten = mx.symbol.Flatten(data=pool)
fc = mx.symbol.FullyConnected(data=flatten, num_hidden=10)
softmax = mx.symbol.SoftmaxOutput(name='softmax',data=fc)
```

Рисунок 3.23 – Модель нейронної мережі, реалізованої на TensorFlow

Час, за який було досягнуто точність 90%: GPU 110 хв; CPU 374 хв.

Caffe. Побудована модель представлена на рисунку 3.26.

```
layer {
 name: "data"
 type: "Input"
 top: "data"
 input_param { shape: { dim: 1
dim: 3 dim: 32 dim: 32 } }
}
layer {
 name: "conv1"
 type: "Convolution"
 bottom: "data"
 top: "conv1"
 param {
  lr_mult: 1
 }
 param {
  lr_mult: 2
 }
 convolution_param {
  num_output: 32
  pad: 2
  kernel_size: 5
  stride: 1
 }
}
layer {
 name: "pool1"
 type: "Pooling"
 bottom: "conv1"
 top: "pool1"
 pooling_param {
  pool: MAX
  kernel_size: 3
  stride: 2
 }
}
layer {
 name: "relu1"
 type: "ReLU"
 bottom: "pool1"
 top: "pool1"
}
layer {
 name: "norm1"
 type: "LRN"
 bottom: "pool1"
 top: "norm1"
 lrn_param {
  local_size: 3
  alpha: 5e-05
  beta: 0.75
  norm_region:
WITHIN_CHANNEL
 }
}
layer {
 name: "conv2"
 type: "Convolution"
 bottom: "norm1"
 top: "conv2"
 param {
  lr_mult: 1
 }
 param {
  lr_mult: 2
 }
 convolution_param {
  num_output: 32
  pad: 2
  kernel_size: 5
  stride: 1
 }
}
layer {
 name: "relu2"
 type: "ReLU"
 bottom: "conv2"
 top: "conv2"
}
layer {
 name: "pool2"
 type: "Pooling"
 bottom: "conv2"
 top: "pool2"
 pooling_param {
  pool: AVE
  kernel_size: 3
  stride: 2
 }
}
layer {
 name: "norm2"
 type: "LRN"
 bottom: "pool2"
 top: "norm2"
 lrn_param {
  local_size: 3
  alpha: 5e-05
  beta: 0.75
  norm_region:
WITHIN_CHANNEL
 }
}
layer {
 name: "conv3"
 type: "Convolution"
 bottom: "norm2"
 top: "conv3"
 convolution_param {
  num_output: 64
  pad: 2
  kernel_size: 5
  stride: 1
 }
}
layer {
 name: "relu3"
 type: "ReLU"
 bottom: "conv3"
 top: "conv3"
}
layer {
 name: "pool3"
 type: "Pooling"
 bottom: "conv3"
 top: "pool3"
 pooling_param {
  pool: AVE
  kernel_size: 3
  stride: 2
 }
}
layer {
 name: "ip1"
 type: "InnerProduct"
 bottom: "pool3"
 top: "ip1"
 param {
  lr_mult: 1
  decay_mult: 250
 }
 param {
  lr_mult: 2
  decay_mult: 0
 }
 inner_product_param {
  num_output: 10
 }
}
layer {
 name: "prob"
 type: "Softmax"
 bottom: "ip1"
 top: "prob"
```

Рисунок 3.26 – Модель нейронної мережі, реалізованої на Caffe

Час, за який було досягнуто точність 90%: GPU 155 хв; CPU 436 хв.

CNTK. Побудована модель представлена на рисунку 3.27.

```
def create_basic_model(input, out_dims):
    with C.layers.default_options(init=C.glorot_uniform(), activation=C.relu):
        net = C.layers.Convolution((5,5), 32, pad=True)(input)
        net = C.layers.MaxPooling((3,3), strides=(2,2))(net)

        net = C.layers.Convolution((5,5), 32, pad=True)(net)
        net = C.layers.MaxPooling((3,3), strides=(2,2))(net)

        net = C.layers.Convolution((5,5), 64, pad=True)(net)
        net = C.layers.MaxPooling((3,3), strides=(2,2))(net)

        net = C.layers.Dense(64)(net)
        net = C.layers.Dense(out_dims, activation=None)(net)

    return net
```

Рисунок 3.27 - Модель нейронної мережі, реалізованої на Caffe

Час, за який було досягнуто точність 90%: GPU 147 хв; CPU 449 хв.

Об'єднаний результат часу тренування представлено на рисунку 3.28.

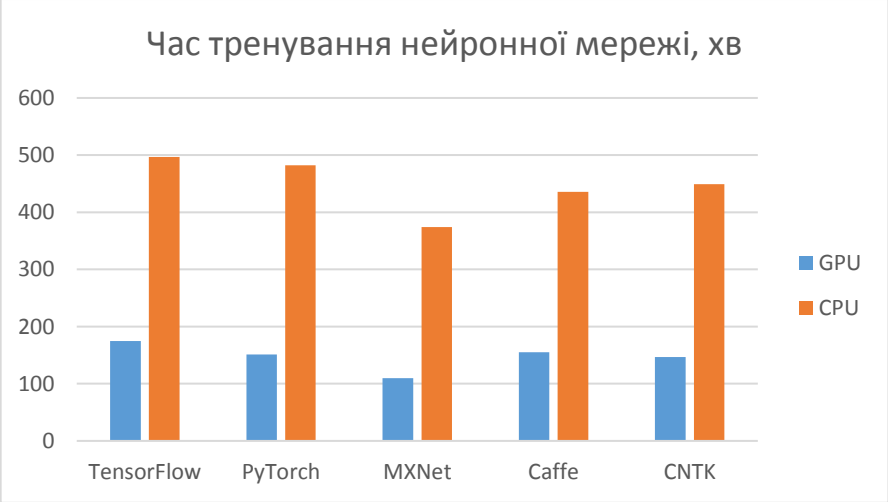


Рисунок 3.28 – Об’єднаний результат тренування

Із отриманого результату можна зробити висновок, що він залежить від замірів, що були проведені для базових операцій, оскільки і в цьому тесті MXNet виявився найшвидшим, а TensorFlow – найповільнішим. Витоку пам’яті не відбулося, оскільки матриці, з якими працював кожен шар були відносно невеликого розміру.

3.8 Висновки за розділом

У даному розділі було проаналізовано існуючі критерії порівняння бібліотек машинного навчання. Загалом, їх оцінюють за можливостями, що вони надають розробнику (візуалізація роботи, паралельне чи розподілене виконання, можливість оптимізації, підтримка попередньо налаштованих моделей тощо), підтримкою фреймворка (наявність документації, форуму запитань-відповідей), часом навчання на різних конфігураціях обчислювальної техніки. В основному, ці критерії ніяк не структуровані, результати за ними доводиться шукати у різних джерелах.

Результати за вже існуючими критеріями було структуровано, було проведено замір часу тренування однієї і тієї ж самої структури нейронної мережі за допомогою обраних бібліотек. До того ж, було проведено замір виконання базових операцій на кожній бібліотеці, та зроблено висновок, що цей час виконання прямо впливання на час тренування. Це означає, що оцінка за критерієм часу виконання базових операцій дозволяє швидше та простіше оцінити час роботи бібліотеки загалом, що значно спрощує підбір бібліотеки для розв’язку конкретної задачі.

4 РОЗРОБЛЕННЯ СТАРТАП ПРОЕКТУ

4.1 Опис ідеї проекту

Ідея проекту полягає в створенні системи, що буде конвертувати обрану архітектуру нейронної мережі відповідно до обраної бібліотеки машинного навчання, та визначати час тренування заданої нейронної, та часу розпізнавання конкретного екземпляру. Розглянемо зміст ідеї, можливі напрямки застосування, основні переваги, які зможе отримати користувач представлено у таблиці 4.1.

Таблиця 4.1 - Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
	1. Оцінка часу роботи на обраній бібліотеці	Можливість оптимізувати час роботи розробленої системи за допомогою застосування нової бібліотек машинного навчання
	2. Конвертація існуючої нейронної мережі відповідно вимог іншої біліотеки	Забезпечує полегшений перехід системи на нову бібліотеку машинного навчання, якщо виникає необхідність змінити реалізацію вже готової системи

Даний проект відрізняється тим, що спеціалізується конвертації та оцінці часу роботи сконвертованої нейронної мережі.

Далі проведено аналіз потенційних техніко-економічних переваг ідеї (чим відрізняється від існуючих аналогів та замінників). Визначено перелік техніко-економічних властивостей та характеристик ідеї.

На ринку наявний конкурент, який надає можливість конвертації нейронних мереж, ONNX [24], що підтримує велику кількість бібліотек машинного навчання.

Проте, використовуючи дану систему, користувач повинен самостійно розгортати її на кожній необхідній машині.

Проведено порівняльний аналіз показників: для власної ідеї визначаються показники, що мають а) гірші значення (W, слабкі); б) аналогічні (N, нейтральні) значення; в) кращі значення (S, сильні) (таблиця 4.2).

Таблиця 4.2 - Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів		W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	ONNX			
1.	Технічна	Конвертація нейронних мереж для бібліотек машинного навчання; Оцінка часу тренування; Оптимізація	Конвертація нейронних мереж у єдиний формат на основі вже розробленої мережі із застосуванням заданої бібліотеки машинного навчання; Оптимізація			Швидкодія; Підтримка багатьох популярних бібліотек
2.	Економічна	Необхідні витрати на обладнання, розробку систему конвертації для підтримки більшості біліотек	Невеликі витрати на обладнання; Не потрібні витрати на розробку системи конвертації	Необхідні витрати на обладнання		
3.	Надійність	Наявність підтримки користувачів; Висока якість послуг; Надійна робота	Надійна робота; Підтримка користувачів		Підтримка користувачів	Можливість розробляти надійні рішення

Визначений перелік слабких, сильних та нейтральних характеристик та властивостей ідеї потенційного товару є підґрунтям для формування його конкурентоспроможності.

4.2 Технологічний аудит ідеї проекту

В межах даного підрозділу необхідно провести аудит технології, за допомогою якої можна реалізувати ідею проекту (технології створення товару).

Визначення технологічної здійсненності ідеї проекту передбачає аналіз таких складових (таблиця 4.3):

Таблиця 4.3 - Технологічна здійсненність ідеї проекту

Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
	Конвертація архітектури нейронної мережі між бібліотеками машинного навчання	Існують аналоги, на які можна орієнтуватися	Технологія доступна
		Середовище розробки PyCharm, мова Python	Технологія доступна
		Середовище розробки Microsoft Visual Studio, мова C++	Технологія доступна
		Використання фізичних машин	Технологія доступна
		Оренда потужностей	Технологія доступна
Обрана технологія реалізації ідеї проекту: в якості середовища розробки було обрано PyCharm та мову Python оскільки для цієї мови біліотеки машинного навчання є найбільш розповсюдженими; для апаратної реалізації обрано оренду потужностей для навчання, щоб не купувати усі необхідні комп’ютери самому.			

За результатами аналізу таблиці робиться висновок щодо можливості технологічної реалізації проекту: так чи ні, а також технологічного шляху, яким це

доцільно зробити (з поміж названих технологій обираються такі, що доступні авторам проекту та є наявними на ринку).

4.3 Аналіз ринкових можливостей запуску стартап-проекту

Визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів.

1) Спочатку проводиться аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку (таблиця 4.4).

Таблиця 4.4 - Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	3
2	Загальний обсяг продаж, грн/ум.од	25 000 за рік
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Відсутні
5	Специфічні вимоги до стандартизації та сертифікації	Сертифікація
6	Середня норма рентабельності в галузі (або по ринку), %	45

Ринок є привабливий для входу та потребує новітніх алгоритмів пошуку найкращих, оптимальних методів вирішення задач.

Надалі визначаються потенційні групи клієнтів, їх характеристики, та формується орієнтовний перелік вимог до товару для кожної групи (таблиці 4.5).

Таблиця 4.5 - Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
Оцінка часу роботи нейронної мережі із застосуванням різних бібліотек машинного навчання	Потенційна група клієнтів є спеціалісти із Data Science	важлива конвертація мережі; важлива оцінка часу роботи	Автоматизація процесу оцінки часу; Підтримка користувачів; Відсутність необхідності встановлення різних бібліотек самому

Після визначення потенційних груп клієнтів проводиться аналіз ринкового середовища: складаються таблиці факторів, що сприяють ринковому впровадженню проекту, та факторів, що йому перешкоджають (таблиці 4.6 та 4.7). Фактори в таблиці подаються в порядку зменшення значущості.

Таблиця 4.6 - Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	З’являються конкуренти	На ринку утворюються конкуренти з підтримкою більшої кількості бібліотек	Удосконалення продукту шляхом підтримки нових бібліотек; Об’єднання із конкурентом
2	Збільшиться вартість обладнання, на якому проводяться тести	Збільшення ціни на графічні процесори, які в основному використовуються для тренування	Проводити оновлення обладнання поступово; Використання дешевих аналогів; Зміна цінової політики

Продовження таблиці 4.6.

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
3	Зменшення попиту	Зменшиться попит на розробку систем, що використовують машинне навчання	Проведення маркетингової кампанії

Таблиця 4.7 - Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Зростання попиту	Зростає попит на оцінку часу роботи нейронної мережі або конвертацію її архітектури	Розширення штату розробників; Пришвидшення розробки
2	Поява нових бібліотек машинного навчання	З’являться нові бібліотеки машинного навчання, які використовують спеціалісти із Data Science	Розширення існуючої системи для підтримки нових бібліотек
3	Об’єднання із існуючими конкурентами	Об’єднати зусилля із існуючими конкурентами для створення більш якісної системи	Купити конкурента або об’єднатись з ним на вигідних умовах

Надалі проводиться аналіз пропозиції: визначаються загальні риси конкуренції на ринку (таблиця 4.8).

Таблиця 4.8 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1.Вказати тип конкуренції - олігополія	Домінує мала кількість розробників подібного програмного забезпечення	Пришвидшення розробки унікального продукту
2. За рівнем конкурентної боротьби - міжнародна	Боротьба ведеться на міжнародному ринку	Збільшення свого впливу на міжнародному ринку
3. За галузевою ознакою - внутрішньогалузева	Боротьба між розробниками систем конвертації нейронних мереж	Покращення якості надання послуг; Підтримка користувачів;
4. Конкуренція за видами товарів: - товарно-видова	Різновиди однієї категорії товару, які здатні задовольнити конкретне бажання покупця	Відсутня
5. За характером конкурентних переваг - нецінова	Ключовим фактором конкурентності є підтримка конвертації між бібліотеками, а не тільки в єдиний формат	Вдосконалення швидкості конвертації
6. За інтенсивністю - не марочна	Не прив’язаний до певної марки, може використовувати будь-де	Співпраця з різними розробниками

Після аналізу конкуренції проводиться більш детальний аналіз умов конкуренції в галузі (таблиця 4.9).

Таблиця 4.9 - Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	Наведені в таблиці 5.2	Можуть з’явитись надаючи більш ширший спектр послуг, а саме більше підтримку бібліотек	Microsoft	Клієнтам потрібні рішення, які дозволяють конвертувати мережу до єдиного формату і між бібліотеками	Частково присутні
Висновки:	Конкуренти не мають усіх можливостей, які плануються у розробці	Конкуренти можуть вийти на ринок	Розроблюють вже готове рішення ONNX	Можна виграти на конвертації мережі між бібліотеками	Часткові рішення присутні на ринку

Робота на ринку можлива, попри наявність конкурентів, оскільки буде надаватись можливість конвертувати нейронні мережі між бібліотеками та оцінювати їх час роботи. Конкурент приводить мережі до єдиного типу, проте не виконує оцінку часу роботи мереж.

На основі аналізу конкуренції, проведеного в таблиця 4.9, а також із урахуванням характеристик ідеї проекту (таблиці 4.2), вимог споживачів до товару (таблиці 4.5) та факторів маркетингового середовища (таблиці № 4.6-4.7) визначається та обґрунтовується перелік факторів конкурентоспроможності. Аналіз оформлюється за таблицею 4.10.

Таблиця 4.10- Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Конвертація мережі між бібліотеками	Дозволяє швидко переробити систему для застосування нової бібліотеки
2	Оцінка часу роботи мережі	Система не тільки конвертує мережі, а й оцінює час роботи із застосуванням нової бібліотеки, що є корисним для кінцевого користувача
3	Підтримка користувачів	Можливість консультувати кінцевих користувачів з метою допомоги їм вибору кращої бібліотеки для розв’язку їх задачі

За визначеними факторами конкурентоспроможності (табл. 4.10) проводиться аналіз сильних та слабких сторін стартап-проекту (табл. 5.11). (С.П. – стартап проект, К.1 – Конкурент 1, К.2 – Конкурент 2)

Таблиця 4.11-Порівняльний аналіз сильних та слабких сторін

№ п/п	Фактор конкурентоспроможності	Бали (1-20)	Рейтинг товарів-конкурентів у порівнянні з … (Конкурент 1,2,3)						
			-3	-2	-1	0	+1	+2	+3
1	Конвертація мережі між бібліотеками	20					К.1		С.П.
2	Оцінка часу роботи мережі	20				К.1			С.П.
3	Підтримка користувачів	18						С.П.	К.1

Фінальним етапом ринкового аналізу можливостей впровадження проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities) (табл. 4.12) на основі виділених ринкових загроз та можливостей, та сильних і слабких сторін (табл. 4.10).

Таблиця 4.12- SWOT- аналіз стартап-проекту

Сильні сторони: конвертація мережі між бібліотеками, оцінка часу роботи	Слабкі сторони: несвоєчасна підтримка бібліотек машинного навчання
Можливості: збільшення попиту	Загрози: зменшення попиту, конкуренція

На основі SWOT-аналізу розробляються альтернативи ринкової поведінки (перелік заходів) для виведення стартап-проекту на ринок та орієнтовний оптимальний час їх ринкової реалізації з огляду на потенційні проекти конкурентів, що можуть бути виведені на ринок (таблиця. 4.9, аналіз потенційних конкурентів).

Визначені альтернативи аналізуються з точки зору строків та ймовірності отримання ресурсів (таблиця 4.13).

Таблиця 4.13- Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Надання консультаційних послуг з розробки нейронних мереж	висока	До одного року
2	Реклама продукту	середня	До трьох місяців
3	Співпраця з великими клієнтами	висока	Необмежено

Після аналізу необхідно зазначити обрану альтернативу.

Основною альтернативою є надання консультаційних послуг з розробки нейронних мереж та вибору бібліотек машинного навчання для їх реалізації.

4.4 Розроблення ринкової стратегії проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (табл. 4.14).

Таблиця 4.14 -Вибір цільових груп потенційних споживачів

Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
Компанії, які співпрацюють зі спеціалістами із Data Science	Споживачі готові сприйняти продукт.	Попит є в цільовому сегменті	Конкуренція існує, залежить від готовності клієнта підвищити витрати	Вхід в сегмент має деякі складності
Навчальні заклади, що спеціалізуються на Data Science	Споживачі готові сприйняти продукт.	Попит є в цільовому сегменті	Конкуренція невелика	Вхід в сегмент не складатиме значних зусиль
Які цільові групи обрано: навчальні заклади, що спеціалізуються на Data Science				

За результатами аналізу потенційних груп споживачів (сегментів) обираються цільові групи, для яких пропонується товар, та визначається стратегія охоплення ринку:

- якщо компанія зосереджується на одному сегменті – вона обирає стратегію концентрованого маркетингу;
- якщо працює із кількома сегментами, розробляючи для них окремо програми ринкового впливу – вона використовує стратегію диференційованого маркетингу;
- якщо компанія працює із всім ринком, пропонуючи стандартизовану програму (включно із характеристиками товару/послуги) – вона використовує масовий маркетинг.

Для роботи в обраних сегментах ринку необхідно сформувати базову стратегію розвитку (табл. 4.15).

Таблиця 4.15 -Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Стратегія наслідування лідера	Стратегія диференціації	Індивідуальний підхід до клієнта; краща якість реалізації системи	Стратегія диференціації передбачає надання товару важливих з точки зору споживача відмітних властивостей, які роблять товар відмінним від товарів конкурентів.

Наступним кроком є вибір стратегії конкурентної поведінки (табл. 4.16).

Таблиця 4.16 - Визначення базової стратегії конкурентної поведінки

Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
Частково	Шукати нових та забирати споживачів у конкурентів	Так, можливість конвертації нейронних мереж	Стратегія заняття конкурентної ніші

На основі вимог споживачів з обраних сегментів до постачальника (стартап-компанії) та до продукту (див. табл. 4.14), а також в залежності від обраної базової стратегії розвитку (табл. 4.15) та стратегії конкурентної поведінки (табл. 4.16) розробляється стратегія позиціонування (табл. 4.17). що полягає у формуванні ринкової позиції (комплексу асоціацій), за яким споживачі мають ідентифікувати торгівельну марку/проект.

Таблиця 4.17 - Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
Конвертація нейронних мереж між бібліотеками	Стратегія спеціалізації	Індивідуальний підхід; Оптимізація	
Оцінка часу роботи нейронної мережі	Стратегія диференціації	Точність виконання	

Результатом виконання підрозділу є узгоджена система рішень щодо ринкової поведінки стартап-компанії, яка визначає напрями роботи стартап-компанії на ринку.

4.5 Розроблення маркетингової програми стартап-проекту

Першим кроком є формування маркетингової концепції товару, який отримає споживач. Для цього у таблиці 4.18 потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Таблиця 4.18 - Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами
1	Конвертація нейронних мереж між бібліотеками	Готові рішення для популярних бібліотек	Висока якість
2	Оцінка часу роботи нейронної мережі на конкретній бібліотеці	Точний і швидкий результат	Можливість вибору бібліотеки для своєї задачі

Надалі розробляється трирівнева маркетингова модель товару: уточнюється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання (табл. 4.19)

Таблиця 4.19 -Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
I. Товар за задумом	Система, що конвертує обрану архітектуру нейронної мережі відповідно до обраної бібліотеки машинного навчання, та визначати час тренування заданої нейронної, та часу розпізнавання конкретного екземпляру		
	Властивості/характеристики	М/Нм	Вр/Тх /Тл/Е/Ор
	1. Конвертація нейронних мереж 2. Оцінка часу виконання		

Продовження таблиці 4.19.

Рівні товару	Сутність та складові
	Якість: Документація, яка допоможе правильно використовувати систему
	Система, яка розгорнута на сервері; сайт, що буде надавати доступ для системи
	Марка: назва компанії
III. Товар із підкріпленням	Програмне забезпечення
Закритий код програми, шифрування	

Наступним кроком є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар (остаточне визначення ціни відбувається під час фінансово-економічного аналізу проекту), яке передбачає аналіз ціни на товари-аналоги або товари субститути, а також аналіз рівня доходів цільової групи споживачів (табл. 4.20). Аналіз проводиться експертним методом.

Таблиця 4.20 -Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
50 тис. грн	75 тис. грн	300 тис. грн	20 - 70 тис. грн

Наступним кроком є визначення оптимальної системи збуту, в межах якого приймається рішення (табл. 4.21):

- проводити збут власними силами або залучати сторонніх посередників (власна або залучена система збуту);
- вибір та обґрунтування оптимальної глибини каналу збуту;
- вибір та обґрунтування виду посередників.

Таблиця 4.21 -Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Вибір послуг на сайті, оплата, постачання послуг	-	Виробник - споживач	Web-сайт

Останньою складової маркетингової програми є розроблення концепції маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування, визначену специфіку поведінки клієнтів (табл. 4.22).

Таблиця 4.22 - Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Знають, які саме послуги треба для вирішення задач	Веб-сайт, телефон, зустрічі	Підтримка клієнтів, індивідуальний підхід	Донесення переваг до клієнтів	Допомога у виборі бібліотеки машинного навчання

Результатом пункту 4.5 є ринкова (маркетингова) програма, що включає в себе концепції послуг, збуту, просування та попередній аналіз можливостей

ціноутворення, спирається на цінності та потреби потенційних клієнтів, конкурентні переваги ідеї, стан та динаміку ринкового середовища, в межах якого буде впроваджено проект, та відповідну обрану альтернативу ринкової поведінки.

Висновки: Є можливість ринкової комерціалізації проекту. Попит на послуги присутні, попри наявність продуктів-аналогів. Розроблений проект має свої переваги над конкурентами у вигляді співвідношення ціна - якість. З огляду на потенційні групи клієнтів є перспективи для входження на ринок. Отже конкурентоспроможність послуг висока. В якості базової стратегії розвитку обрана стратегія диференціації. В якості альтернативи можливе надання консультаційних послуг щодо вибори бібліотек машинного навчання.

ВИСНОВКИ

Під час написання магістерської дисертації були розглянуті найпопулярніші бібліотеки машинного навчання на прикладі розв'язку задачі розпізнавання. Детально розглянуто тему машинного навчання, задачу розпізнавання та нейронні мережі. Проведено порівняння на згортковій нейронній мережі для розпізнавання зображень.

Для оцінки обрано найпопулярніші на сьогодні бібліотеки машинного навчання: TensorFlow, PyTorch, MXNet, CNTK, Caffe.

Проаналізовано вже існуючі критерії порівняння та результати цих порівнянь, в результаті чого було виявлено неповноту цих критеріїв оцінки. Головним доповненням до існуючих критеріїв була оцінка часу виконання базових операцій для нейронних мереж, оскільки результатів проведення за таким критерієм не було знайдено. Заміри для кожного з обраних фреймворків проводилися за загальними критеріями (як будується архітектура мережі, наявність попередньо натренованих моделей, наявність прикладів тощо) та часовими критеріями (час виконання базових операцій, час тренування нейронної мережі до заданої точності). Зроблено висновок, що час тренування нейронної мережі залежить від часу виконання базових операцій.

В результаті оцінки кожного фреймворку, за загальними критеріями було отримано, що MXNet є найкращими фреймворком, а Caffe – найгіршим. MXNet є наймолодшим представником, включає у себе необхідні для розробника компоненти, швидко розвивається, тому і отримав найкращий результат. Caffe навпаки, хоч і використовується досі, є застарілим фреймворком, тому деякі моменти його реалізації не є актуальними.

Кожен розробник особисто обирає важливість того чи іншого критерію, в залежності від умов поставленої задачі, обчислювальних можливостей, необхідності часто змінювати і перенавчати архітектуру нейронної мережі. Якщо було обрано техніку навчання fine-tuning, то не варто обирати TensorFlow, адже це не основна задача цього фреймворку, в цьому випадку Caffe має перевагу. Якщо важлива швидкість навчання і переналаштування мережі, то краще звернути увагу на MXNet та PyTorch, адже вони можуть забезпечити цю швидкість. Якщо задача складна, не

вистачає готових рішень, потрібна підтримка професіоналів у сфері машинного навчання, то можна обрати TensorFlow як найрозповсюдженіший фреймворк, що означає, що легше буде отримати певну допомогу при вирішенні задачі.

За результатами тесту щодо часу виконання було визначено, що найшвидшим виявився фреймворк MXNet. Варто відмітити, що саме символічний підхід працював швидше, оскільки під час такого підходу у пам’яті зберігаються лише ті дані, які необхідні, а не абсолютно усі обчислення. TensorFlow виявився найповільнішим, а імперативні підходи у CNTK і PyTorch на малих розмірностях задач виявилися швидшими, в деяких випадках PyTorch був навіть швидший за імперативний підхід у MXNet.

Розроблено стартап проект «Система, що конвертує обрану архітектуру нейронної мережі відповідно до обраної бібліотеки машинного навчання, та визначає час тренування заданої нейронної, та часу розпізнавання конкретного екземпляру».

ПЕРЕЛІК ПОСИЛАНЬ

1. Pedro Domingos A Few Useful Things to Know about Machine Learning/ A. Paszke // Department of Computer Science and Engineering University of Washington – Seattle, WA 98195-2350, U.S.A. – P. 1 - 3

2. CS 295: Pattern Recognition, Course Notes / Snapp R // Department of Computer Science, University of Vermont – P. 3 – 7

3. Pattern Classification. (2nd ed.). / . Duda, R.O., Hart, P.E., and Stork D.G// New York: Wiley-Interscience Publication – 2001 – P. 12- 16.

4. Intelligent Sensor Systems, Course Notes / Gutierrez-Osuna R // Department of Computer Science, Wright State University – P. 22 - 26

5. Celebi Tutorial: Neural Networks and Pattern Recognition Using MATLAB [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.byclb.com/TR/Tutorials/neural_networks/ch1_1.htm (дата звернення 10.05.2018). – Назва з екрана.

6. Generalized Feature Extraction for Structural Pattern Recognition in TimeSeries Data / Olszewski R. T.// PhD. Thesis at School of Computer Science - Carnegie Mellon University, Pittsburgh. – P. 12 – 19

7. Convolutional Neural Networks (CNNs / ConvNets) [Електронний ресурс]:[Веб-сайт] – Режим доступу: http://cs231n.github.io/neural-networks-1/ (дата звернення 10.05.2018) – Назва з екрана.

8. Convolutional Neural Networks (CNNs / ConvNets) [Електронний ресурс]:[Веб-сайт] – Режим доступу: http://cs231n.github.io/convolutional-networks/ (дата звернення 10.05.2018) – Назва з екрана.

9. Deep Learning Programming Style [Електронний ресурс] : [Веб-сайт]. – Режим доступу:https://mxnet.incubator.apache.org/architecture/program_model.html (дата звернення 10.05.2018). – Назва з екрана.

10. TensorFlow: A System for Large-Scale Machine Learning / Martín Abadi, Paul Barham // 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16) - November 2–4, 2016 - Savannah, GA, USA – P. 266 - 270

11. MXNet: A Flexible and Efficient Machine Learning for Heterogeneous Distributed Systems / Tianqi Chen, Mu L – P. 2 - 4

12. The Microsoft Cognitive Toolkit [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://docs.microsoft.com/en-us/cognitive-toolkit/ (дата звернення 10.05.2018). – Назва з екрана.

13. PyTorch [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://pytorch.org/about/ (дата звернення 10.05.2018). – Назва з екрана.

14. Caffe Deep learning framework by BAIR [Електронний ресурс] : [Веб-сайт]. – Режим доступу: http://caffe.berkeleyvision.org/ (дата звернення 10.05.2018). – Назва з екрана.

15. The CIFAR-10 dataset [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.cs.toronto.edu/~kriz/cifar.html (дата звернення 10.05.2018). – Назва з екрана.

16. Machine Learning Frameworks Comparison [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://blog.paperspace.com/which-ml-framework-should-i-use/ (дата звернення 10.05.2018). – Назва з екрана.

17. Battle of the Deep Learning frameworks—Part I: 2017, even more frameworks and interfaces [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://towardsdatascience.com/battle-of-the-deep-learning-frameworks-part-i-cff0e3841750 (дата звернення 10.05.2018). – Назва з екрана.

18. Choosing a Machine Learning Framework in 2018 [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://agi.io/2018/02/09/survey-machine-learning-frameworks/ (дата звернення 10.05.2018). – Назва з екрана.

19. Comparing Deep Learning Frameworks: A Rosetta Stone Approach [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.kdnuggets.com/2018/03/deep-learning-frameworks.html (дата звернення 10.05.2018). – Назва з екрана.

20. DAWNBench An End-to-End Deep Learning Benchmark and Competition [Електронний ресурс] : [Веб-сайт]. – Режим доступу: http://dawn.cs.stanford.edu/benchmark/#cifar10 (дата звернення 10.05.2018). – Назва з екрана.

21. CUDA Zone | NVIDIA Developer [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://developer.nvidia.com/cuda-zone (дата звернення 10.05.2018). – Назва з екрана.

22. Deep Learning Part 1: Comparison of Symbolic Deep Learning Frameworks [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.r-bloggers.com/deep-learning-part-1-comparison-of-symbolic-deep-learning-frameworks/ (дата звернення 10.05.2018). – Назва з екрана.

23. Grad backward memory leak? [Електронний ресурс]: [Веб-сайт]. – Режим доступу: https://github.com/pytorch/pytorch/issues/3824 (дата звернення 10.05.2018). – Назва з екрана.

24. ONNX: Open neural network exchange format [Електронний ресурс]: [Веб-сайт]. – Режим доступу: https://onnx.ai/ (дата звернення 10.05.2018). – Назва з екрана.

Додаток А – Тези доповіді «Порівняння фреймворків машинного навчання» на VI Міжнародній науково-практичній конференції «Summer InfoCom Advanced Solutions 2018»

Порівняння фреймворків машинного навчання

Дорогий Ярослав
КПІ ім. Ігоря Сікорського
Київ, Україна
cisco.rna@gmail.com

Левченко Ксенія
КПІ ім. Ігоря Сікорського
Київ, Україна
ksulevchenko95@gmail.com

Анотація. **Тези доповіді містять порівняння існуючих фреймворків машинного для мови Python. В роботі коротко описано переваги та недоліки обраних фреймворків. Авторами були дані рекомендації щодо використання кожного з фреймворків.**

Ключові слова: розпізнавання зображень, машинне навчання, згорткові нейронні мережі, фреймворки машинного навчання.

Вступ

У наш час є актуальною проблема побудови нейронних мереж, адже вони використовуються у таких галузях як медична діагностика (виявлення вад на рентгенівських знімках), розпізнавання текстів, мови, технічна діагностика, системи контролю.

Нейронні мережі дозволяють розв'язувати складні задачі класифікації, у яких без автоматизації було б задіяно велику кількість людських ресурсів. Це такі галузі як медична діагностика (виявлення вад на рентгенівських знімках), розпізнавання текстів, мови, технічна діагности-ка, системи контролю.

Для побудови схожої власної системи (або для початку побудови таких систем) дуже зручно використовувати вже готові рішення, які значно понижують поріг входження у галузь розробки систем розпізнавання. На сьогодні існує велика кількість фреймворків, що полегшують реалізацію системи на основі нейронних мереж та дозволяють сфокусуватись на архітектурі системи, на задачах, що вона розв'язує, а не на розробці конкретного програмного забезпечення.

Постановка задачі

Спочатку визначимо задачу, на прикладі якої буде проводитись порівняння найпопулярніших фреймворків машинного навчання. Нехай, нам необхідно побудувати систему розпізнавання певних об'єктів на зображеннях. Оберемо такі об'єкти, що присутні у публічних датасетах, наприклад об'єкти з CIFAR-10 датасету [1]. Оскільки це задача розпізнавання зображень, то для своєї системи оберемо convolutional neural network (згорткову нейронну мережу) [2], що найкраще справляється з таким типом задач через особливості своєї архітектури. Мову програмування обрано Python, оскільки для неї фреймворки представлені найширше.

Виходячи з постановки нашої задачі можна виділити такі критерії порівняння: простота опису архітектури нейронної мережі, підтримка роботи із згортковими нейронними мережами, простота переналаштування архітектури мережі, що є дуже важливим фактором, оскільки швидке переналаштування мережі дозволяє легко проводити експерименти з різними варіантами нейронних мереж можливість навчання на графічному процесорі (а саме підтримка платформи CUDA [3]), доступність прикладів, підтримка попередньо натренованих моделей, які дозволяють покращити результати навчання, підтримка різних ОС.

Порівняння фреймворків

Для порівняння було обрано такі фреймворки: PyTorch [4], TensorFlow [5], Caffe [6], MxNet [7].

PyTorch. Це фреймворк, який дозволяє проводити обчислення швидко на графічному процесорі та будувати нейронні мережі. Переваги: підтримка попередньо натренованих моделей, швидка зміна архітектури нейронної мережі

Таблиця 1

Порівняння фреймворків за обраними критеріями

Фреймворк	Архітектура	Переналаштування	Підтримка попередньо натренованих моделей	Наявність прикладів	Тренування	ОС	Паралельні обчислення
PyTorch	Методи, що мають назву шарів CNN	Auto-differentiation	Присутня велика кількість моделей, легке тренування	Представлені на сайті фреймворку, репозиторії на github	Треба писати самому	Windows, Linux	+
TensorFlow	Методи, що мають назву шарів CNN	Auto-differentiation	Не так розповсюджено	Представлені на сайті фреймворку, репозиторії на github, велика кількість окремих статей	Готові методи	Windows, Linux	+

CAFFE	Методи, що мають назву шарів CNN; JSON структура	Модель має бути перенатренована	Фреймворк націлений саме на fine-tuning	Репозиторії на github	Не потрібен код, тільки налаштування	Linux, формальна підтримка Windows	-
MXNet	Методи, що мають назву шарів CNN	Auto-differentiation	Присутня велика кількість моделей, легке тренування	Представлені на сайті фреймворку, репозиторії на github	Готові методи	Windows, Linux	+

Таблиця 2

Оцінка кожного фреймворку за обраними критеріями

Фреймворк	Архітектура	Переналаштування	Підтримка попередньо натренованих моделей	Наявність прикладів	Тренування	ОС	Паралельні обчислення	Сума (max = 35)
PyTorch	4.5	5	4	4	3	5	5	30.5
TensorFlow	4.5	5	3	5	5	5	5	32.5
CAFFE	5	0	5	3.5	5	4	0	22.5
MXNet	4.5	5	4.5	4	5	5	5	33

завдяки техніці reverse-mode auto-differentiation (автоматичне диференціювання з рухом назад), підтримка візуалізації побудованої архітектури, опис моделі за допомогою методів, що мають назви шарів згорткової нейронної мережі, підтримка візуалізації побудованої архітектури, опис моделі за допомогою методів, що мають назви шарів згорткової нейронної мережі (conv, relu, max_pool), що пришвидшує початок роботи з цим фреймворком, підтримка CUDA. До недоліків можна віднести те, що код для тренування необхідно писати самому, що сповільнює розробку [8].

TensorFlow. Фреймворк від Google, який замінив Theano, став простішим і швидшим на відміну від попередника, є найпопулярнішим серед розробників. Переваги: архітектура мережі описується у термінах, що використовуються у CNN, має детальну візуалізацію TensorBoard, що будує граф під час навчання, готова реалізація тренування, використовує auto-differentiation для переналаштування архітектури, підтримує паралельні обчислення, підтримка CUDA. Недоліки: значно повільніший за інші фреймворки (наприклад, MxNet), невелика кількість попередньо натренованих моделей, складний для початківця [9].

Caffe. Фреймворк, який спеціалізується саме на розпізнаванні зображень. Переваги: гарна підтримка CNN, велика кількість попередньо натренованих моделей, опис моделей відбувається за допомогою JSON, що дозволяє краще розуміти вкладеність та структуру шарів без візуалізації, або за допомогою вбудованих методів, присутня візуалізація, типи шарів описуються у відомих термінах для CNN, тренування проводиться без написання додаткового коду, лише за допомогою файлів налаштувань, підтримка CUDA. Недоліки: важко розширювати для виконання інших задач розпізнавання повільно працює з великими мережами, які не були попередньо натреновані, відсутність швидкого переналаштування, навчання проводитиметься спочатку, майже відсутня підтримка [10].

MXNet. Фреймворк для машинного навчання від Microsoft і Amazon. Переваги: набагато швидший за інші фреймворки (швидше навчання, менше використання оперативної пам'яті), підтримує попередньо натреновані моделі (у прикладах пристуній готовий код для fine-tuning), опис моделі проводиться у термінах CNN, присутня візуалізація, можливе швидке переналаштування завдяки auto-differentiation, готовий код для тренування, підтримка CUDA. Недоліки: необхідно встановлювати окремі версії фреймовпка для GPU та CPU, невелика кількість документації [11].

Основні критерії порівняння винесені в табл. 1. Варто додати, що усі фреймворки підтримують навчання на GPU за допомогою CUDA, та мають візуалізацію побудованої мережі.

В результаті порівняння кожного фреймворку за обраними критеріями, їх було оцінено по калі від 0 до 5 (від найгіршого результату до найкращого). Результат цієї оцінки представлено в табл. 2.

Кожен критерій також було оцінено по шкалі від 0 до 5 (від зовсім не важливого, до дуже важливого) з точки зору важливості для прийняття рішення, який фреймворк використовувати. Результати представлено в табл. 3.

Таблиця 3

Оцінка критеріїв за їх важливістю

Критерій	Оцінка (0-5)
Архітектура	5
Переналаштування	5
Підтримка попередньо натренованих моделей	3
Наявність прикладів	3

Тренування	4
ОС	2
Паралельні обчислення	4

Висновки

Порівняння, що приведено вище полегшує попередній вибір фреймворку для розв'язку задач, які є подібними до поставленої. В результаті оцінки кожного фреймворку, за обраними критеріями було отримано, що MXNet є найкращими фреймворком, а Caffe – найгіршим. MXNet є наймолодшим представником, включає у себе необхідні для розробника компоненти, швидко розвивається, тому і отримав найкращий результат. Caffe навпаки, хоч і використовується досі, є застарілим фреймворком, тому деякі моменти його реалізації не є актуальними.

Звичайно, що кожен розробник особисто обирає важливість того чи іншого критерію, в залежності від умов поставленої задачі, обчислювальних можливостей, необхідності часто змінювати і перенавчати архітектуру нейронної мережі. Якщо було обрано техніку навчання fine-tuning, то не варто обирати TensorFlow, адже це не основна задача цього фреймворку, в цьому випадку Caffe має перевагу. Якщо важлива швидкість навчання і переналаштування мережі, то краще звернути увагу на MxNet та PyTorch, адже вони можуть забезпечити цю швидкість. Якщо задача складна, не вистачає готових рішень, потрібна підтримка професіоналів у сфері машинного навчання, то можна обрати TensorFlow як найрозповсюдженіший фреймворк, що означає, що легше буде отримати певну допомогу при вирішенні задачі.

Література

1. CIFAR-10 and CIFAR-100 datasets [Електронний ресурс]: [Веб-сайт]. – Режим доступу: https://www.cs.toronto.edu/~kriz/cifar.html (дата звернення 30.04.2018). – Назва з екрана.
2. CS231n: Convolutional Neural Networks for Visual Recognition [Електронний ресурс] : [Веб-сайт]. – Режим доступу: http://cs231n.github.io/convolutional-networks/ (дата звернення 30.04.2018). – Назва з екрана.
3. CUDA Zone | NVIDIA Developer [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://developer.nvidia.com/cuda-zone (дата звернення 30.04.2018). – Назва з екрана.
4. PyTorch [Електронний ресурс] : [Веб-сайт]. – Режим доступу: http://pytorch.org/ (дата звернення 30.04.2018). – Назва з екрана.
5. TensorFlow [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.tensorflow.org/ (дата звернення 30.04.2018). – Назва з екрана.
6. Caffe | Deep Learning Framework [Електронний ресурс] : [Веб-сайт]. – Режим доступу: http://caffe.berkeleyvision.org/ (дата звернення 30.04.2018). – Назва з екрана.
7. MXNet: A scalable deep learning framework [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://mxnet.incubator.apache.org/ (дата звернення 30.04.2018). – Назва з екрана.
8. Adam Paszke Automatic differentiation in PyTorch/ A. Paszke, S. Gross // 31st Conference on Neural Information Processing Systems (NIPS 2017) – Long Beach, CA, USA, 2017. – С. 1 -3
9. Martin Abadi TensorFlow: A system for large-scale machine learning/ M. Adabi, P. Barham // 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association (2016) – SAVANNAH, GA, USA, 2016. – P. 1 - 5.
10. Yangqing Jia Caffe: Convolutional Architecture for Fast Feature Embedding/ Y. Jia, E. Shelhamer // ACM MULTIMEDIA 2014 OPEN SOURCE SOFTWARE COMPETITION – 2014 – P. 2 – 3.
11. Tianqi Chen MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems/ T. Chen, M. Li // In Neural Information Processing Systems, Workshop on Machine Learning Systems – 2016 – P. 2 – 5.

Додаток Б – Стаття «Порівняння часу виконання базових операцій фреймворків машинного навчання» у журналі Інфокомунікаційні системи та технології № 2(2)

Порівняння часу виконання базових операцій фреймворків машинного навчання

Дорогий Ярослав
КПІ ім. Ігоря Сікорського
Київ, Україна
cisco.rna@gmail.com

Левченко Ксенія
КПІ ім. Ігоря Сікорського
Київ, Україна
ksulevchenko95@gmail.com

Анотація. **Стаття містить дослідження порівняння часу виконання базових операцій фреймворків машинного навчання та пам'яті, що виділялась. В роботі приведено результати експериментів для обраних операцій і розмірностей задач. Авторами були проаналізовані отримані результати та обрано стратегію розвитку представленої ідеї.**

Ключові слова: машинне навчання, нейронні мережі, фреймворки машинного навчання, імперативний і символічний підхід.

Вступ

Проблема побудови нейронних мереж є актуальною, оскільки вони використовуються в системах, де необхідне розпізнавання чи класифікація об'єктів без участі людини.

На сьогодні існує безліч фреймворків для побудови, навчання нейронних мереж. Дуже важливим для побудови такої системи є правильний вибір фреймворка для реалізації, оскільки швидкість його роботи прямо впливає на швидкість розробки.

Зазвичай порівняння проводиться за критерієм часу, за який мережа досягне певної точності. Проте для такого порівняння потрібно будувати мережу для кожного фреймворку окремо, тому її робота може відрізнятись. У даній роботі порівнюються фреймворки за часом виконання базових операцій, які використовуються під час тренування нейронної мережі.

Постановка задачі

Для порівняння було обрано ті базові операції, які найчастіше використовуються під час тренування нейронної мережі, а саме:

- сума і множення матриць. Використовуються під час перемноження входів (*inputs*) та ваг (*weigths*), суми ваг та зміщення (*bias*), операції згортки (*convolution*) і т.д. Матриці можуть бути довільної розмірності.

- транспонування матриць. Оскільки в нейронній мережі необхідно обробляти ваги та входи довільної розмірності, то ці матриці можуть не задовольняти правилам множення. Саме тому матриці доводиться часто транспонувати.

Зі схеми нейрона, що представлено на рис. 1 видно, що у ньому використовуються активаційні функції (activation function).

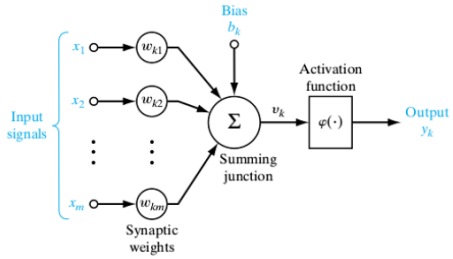


Рис. 1. Схема штучного нейрона

Серед активаційних функцій було обрано ReLU (rectified linear unit), яка представлена у (1) та Sigmoid, яка представлена у (2).

$$f(x) = max(0, x) \quad (1)$$

$$f(x) = 1 / (1 + e^{-x}) \quad (2)$$

Для кожної епохи навчання розраховують функцію втрат (loss function) – функцію, яку мінімізують алгоритми машинного навчання. За допомогою функції втрат оцінюють якість тренування мережі. Одними з реалізацій функції втрат є reduce sum, що обраховує суму усіх елементів матриці за заданими або усіма осями, або reduce mean – визначення середнього значення послідовності.

Заміри проводились на CPU Intel Core i7-3517U 2.4 GHz, 8 GB RAM, OS Windows 10. Тестові скрипти були виконані на мові програмування Python 3.6.4.

Імперативні та символічні підходи

Для порівняння було обрано популярні такі фреймворки: Tensorflow[1], MXNet [2], CNTK [3], PyTorch [4] – сим-волічні фреймворки (symbolic frameworks).

Різниця між імперативними (imperative) і символічними фреймворками полягає у способі виконання операцій. Імперативний підхід виконує операцію саме в той час, коли її викликають. Це означає, що коли виконується рядок $c = b * a$ з лістингу на рис. 2, то програма виконує обчислення значень, що були вказані вище.

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
```

Рис. 2. Лістинг імперативного підходу програмування

Символічний підхід відрізняється тим, що спочатку визначається функція, яка буде виконуватись, проте без конкретних значень. Ця функція має змінні, які потім будуть ініціалізуватись

вже реальними значеннями. Тоді необхідно скомпілювати функцію та виконати її вже із реальними значеннями.

```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```

Рис. 3. Лістинг символічного підходу програмування

Під час виконання рядку $c = b * a$ ніяких обчислень не відбувається. Замість цього ця операція генерує граф обчислень (computation graph), який представляє це обчислення. Приклад графу, який буде побудовано під час виконання рядка $c = b * a$ представлено на рис. 4.

Важливим моментом символічному підходу є те, що програма явно або неявно містить крок компіляції. Компіляція перетворює граф обчислень в функцію, яка буде виконана пізніше. У лістингу, що зображено на рис. 3,

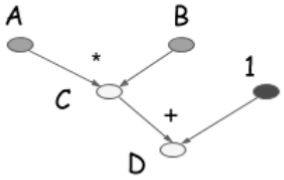


Рис. 4. Граф обчислень, згенерований програмою

фактичне обчислення відбудеться під час виконання останнього рядку. Символічний підхід визначає чітке розділення між створенням графа обчислень і його виконанням. Для нейронних мереж зазвичай усю модель визначають як єдиний граф обчислень.

Імперативний підхід більш інтуїтивно зрозумілий, гнучкий. Будь-який алгоритм з імперативним підходом пишеться «as is», майже природньою мовою.

Символічний підхід є більш ефективним за пам'яттю та швидкістю. Для виконання обчислення «тут і зараз» необхідно виділяти пам'ять під кожний рядок програми. Коли ми будуємо граф обчислень і знаємо, що в результаті нам потрібно тільки значення *d* з рисунку 4, то стає можливим перевикористати пам'ять, що була виділена для зберігання проміжних значень. Для імперативного підходу ми можемо очищати пам'ять, наприклад видаляти з пам'яті біти, виділені для *b*, щоб зберегти *c*, потім видалити з пам'яті *c*, щоб зберегти *d*.

Символічний підхід є більш обмеженим. Коли відбувається виклик компіляції графу обчислень, то система знає, що тільки значення *d*, є необхідним. Проміжні значення є невидимими для нас.

Символічний підхід може безпечно перевикористовувати пам'ять, яку було виділено для обчислень «тут і зараз». Проте при цьому ми втрачаємо доступ до *c*, тому імперативний підхід тут має більшу перевагу, оскільки ми можемо отримати значення будь-якої змінної у потрібний для нас час.

Символічний підхід може використовувати інший спосіб оптимізації, який називається складання (folding) (рис. 5). Це означає, що, наприклад, операція множення і додавання може бути об'єднана в одну операцію. Якщо обчислення проводяться на GPU, то тільки одне ядро GPU буде використано для цієї операції замість двох. Така оптимізація збільшує швидкість обчислень.

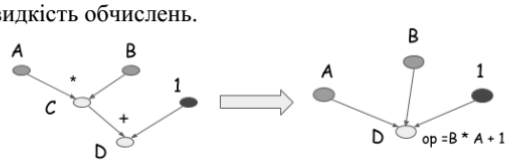


Рис. 5. Операція складання

Операція складання неможлива для імперативного підходу, оскільки проміжні значення можуть бути використані в майбутньому. Операція складання можлива для символічного підходу, оскільки ми отримуємо повний граф обчислень та чітко розуміємо, які значення нам потрібні, а які ні [5].

Під час проведення замірів було окремо порівняно швидкість операцій для імперативного та символічного підходу на основі фреймворка MXNet.

Порівняння фреймворків

Кожен з фреймворків було порівняно за швидкістю виконання та пам'яттю, яку вони використовували під час виконання операцій. Конкретні результати тестів представлено після опису кожного з них.

Tensorflow. Це символічний фреймворк з відкритим кодом для проведення обчислень з використанням графа обчислень та машинного навчання від Google Brain Team. Використовує виключно символічний підхід. На сьогодні є найпопулярнішим фреймворком, що використовується на багатьох проектах, хоча і створювати системи на ньому доволі важко через складний API.

Швидкість: із зростанням розмірності задачі, Tensorflow виконував операції все повільніше і повільніше, багато часу уходило на компіляцію (ініціалізацію сесії). Проте і на малій розмірності Tensorflow показав найгірші результати. Про повільне виконання зауважують різні ентузіасти, що порівнювали швидкість тренування для Tensorflow [6], [7].

Пам'ять: під час виконання операцій було помітно, що пам'ять дещо зростала (300 MB – 500 MB – 800 MB – 1100 MB), а потім знову падала до початкового значення у 300 MB. Це означає, що Tensorflow звільняє пам'ять під час виконання операцій.

Лістинг для тесту Tensorflow представлено на рис. 6.

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for j in range(1000):
        start_time = time.time()
        sess.run(operation) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рис. 6. Лістинг для тесту Tensorflow

MXNet. Це фреймворк від Apache, який підтримує і символічний, і імперативний підходи. Гнучкий та швидкий фреймворк, який підтримується Microsoft і Amazon. Набирає популярність серед спеціалістів, що працюють у сфері машинного навчання.

Швидкість: із збільшенням розмірності задачі час виконання операцій на MXNet майже не змінювався, був відносно малим. У порівнянні між символічним та імперативним підходами символічний підхід показав результати в середньому у 4 рази кращі. Показав найкращу швидкість виконання серед усіх.

Пам'ять: що для імперативного, що для символічного підходів пам'ять зростала не сильно (300 MB – 450 MB) та швидко поверталась до початкового значення, тобто MXNet також звільняє пам'ять.

Лістинг для тесту MXNet представлено на рис. 7, 8.

```
def perform_operation(shape, mul):
    time_sum = 0
    a = mx.symbol.Variable('A')
    e = operation(a) # assign operation
    for j in range(1000):
        a_data = mx.nd.uniform(low=-deviation,
                               high=deviation, shape=[shape] * mul)
        executor = e.bind(mx.cpu(), {'A':a_data})
        start_time = time.time()
        executor.forward() # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рис. 7. Лістинг для тесту MXNet (символічний підхід)

```
def operation(shape, mul):
    time_sum = 0
    for j in range(1000):
        a = mx.nd.uniform(low=-deviation, high=deviation,
                          shape=[shape] * mul)
        start_time = time.time()
        mx.operation(a) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рис. 8. Лістинг для тесту MXNet (імперативний підхід)

PyTorch. Символічний фреймворк, Python версія фреймворку, був розроблений Facebook. Налаштований для проведення обчислень на GPU.

Швидкість: перевірявся тільки імперативний підхід, тому швидкість виконання зростала із збільшенням розмірності задачі, швидкість сильно зростала для множення матриць, обчислення Sigmoid та Reduce Sum. Транспонування, сума матриць і виконання функції ReLU були швидкими незалежно від розмірності задачі.

Пам'ять: через імперативний підхід із збільшенням задачі до розмірності чотиривимірних матриць відбувся витік пам'яті, що унеможливило перевірку на великих даних. Для малої розмірності задачі пам'ять трималась на рівні 400-500 MB. Проте про витік пам'яті є свідчення і на інших операціях, в тому числі і на GPU [8].

Лістинг для тесту PyTorch представлено на рис. 9.

```
def perform_operation(shape, mul):
    time_sum = 0
    a = torch.randn([shape]*mul, dtype=torch.double)
    for j in range(1000):
        start_time = time.time()
        torch.operation(a) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рис. 9. Лістинг для тесту PyTorch

CNTK. Microsoft Cognitive Toolkit (або CNTK) - фреймворк для машинного навчання від Microsoft, має покращені алгоритми для обробки великих датасетів. Не для всіх операцій був представлений API, тому сума і транспонування матриць не представлені для цього фреймворку.

Швидкість: був використаний імперативний підхід, тому швидкість зростала із збільшенням розмірності задачі, проте в основному швидше, ніж PyTorch і набагато швидший за Tensorflow.

Пам'ять: через імперативний підхід пам'ять із збільшенням задачі не вивільнялась, тому це унеможливило провести заміри для чотиривимірних матриць через витік пам'яті. При виконанні менших задач пам'ять трималась на рівні 600 MB – 800 MB.

Лістинг для тесту CNTK представлено на рис. 10.

```
def perform_operation(shape, mul):
    time_sum = 0
    for j in range(1000):
        a = np.random.sample(size = [shape]*mul)
        start_time = time.time()
        cntk.operation(a) # operation execution
        end_time = time.time()
        time_sum += end_time - start_time
```

Рис. 10. Лістинг для тесту CNTK

Результати замірів

Тестування проводилось для матриць різних розмірностей: 10×10, 10×10×10, 10×10×10×10, 100×100, 100×100×100, 100×100×100×100, 1000×1000. Умовні позначення на кожному з графіків: «Add matrices (…)» - сума двох матриць, «Mul 2 matrcies (…)» - множення двох матриць, «Mean» - reduce mean - пошук середнього значення за усіма осями, «Relu» - обчислення значення функції ReLU, «Sigmoid» - обчислення значення функції Sigmoid, «Reduce sum» - виконання операції reduce sum, «Transpose» - транспонування матриці. По вертикальній осі вказано час у секундах.

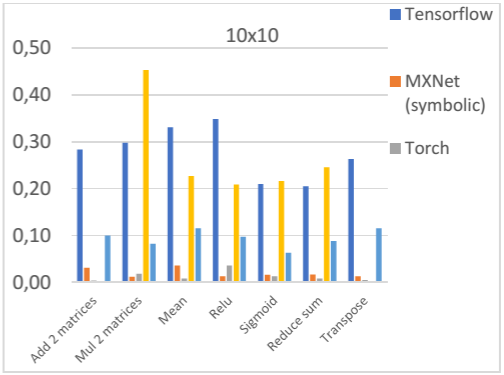


Рис. 11. Тест для матриці 10×10

З рис. 11 видно, що вже на малих розмірностях MXNet працює швидше за інші фреймворки, проте Torch для операцій Reduce Sum, Mean та суми двох матриць і транспонування був швидшим, майже всюди найгірші результати показує Tensorflow, найгірше множення у CNTK.

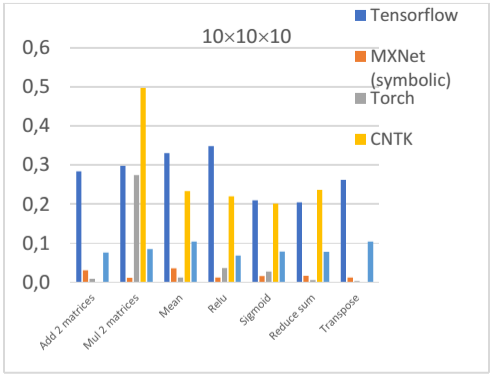


Рис. 12. Тест для матриці 10×10×10

З рис. 12 видно, що MXNet залишається майже у всіх варіантах найшвидшим, за винятками для PyTorch, що були описані вище. В основному, результат схожий на попередній.

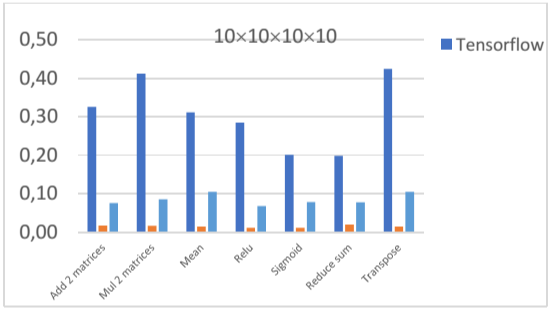


Рис. 13. Тест для матриці 10×10×10×10

З рис. 13 видно, що результати для трьох представлених тестів схожі на попередній, проте для CNTK і PyTorch вже на такій розмірності відбувся витік пам'яті.

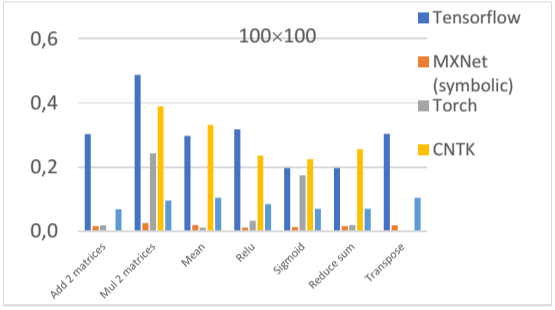


Рис. 14. Тест для матриці 100×100

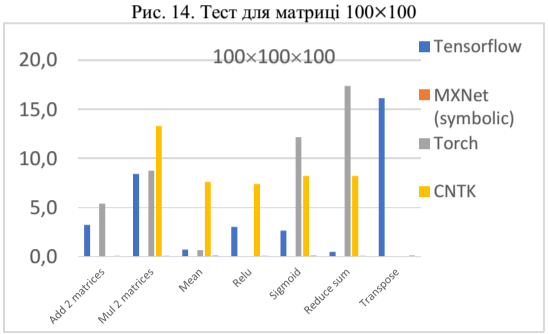


Рис. 15. Тест для матриці 100×100×100

З рис. 14 і 15 видно, що MXNet на цих розмірностях вже набагато швидший за інші фреймворки, окрім операції транспонування, тут PyTorch кращий. Tensorflow поступово сповільнюється.

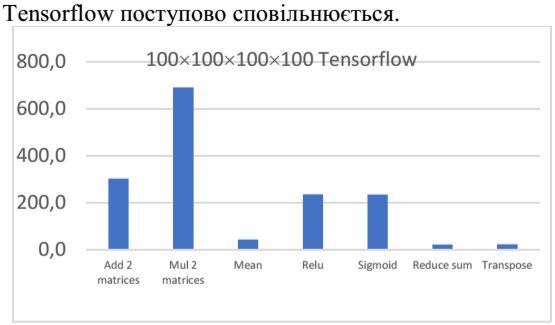


Рис. 16. Тест для матриці 100×100×100×100 для TensorFlow

З рис. 16 видно, що Tensorflow вже дуже сильно сповільнюється, у порівнянні з MXNet. Для CNTK і PyTorch на цій розмірності стався витік пам'яті.

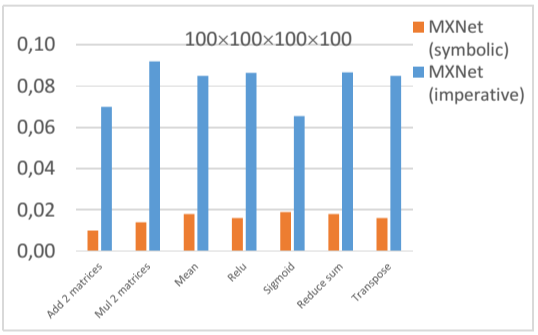


Рис. 17. Тест для матриці 100×100×100×100 для MXNet

З рис. 17 видно, що MXNet стабільно швидко виконує операції для більшої розмірності, при чому символічний підхід набагато швидший за імперативний.

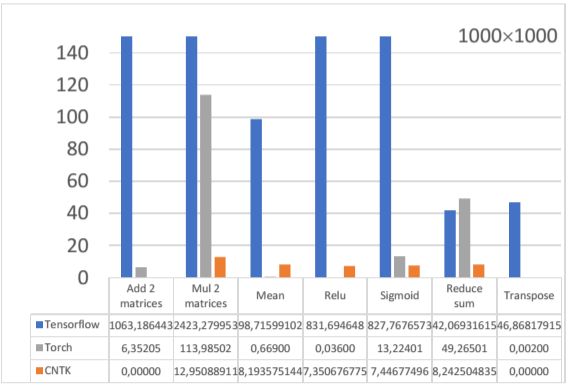


Рис. 18. Тест для матриці 1000×1000

З рис. 18 видно, що Tensorflow знову показує найгірші результати, PyTorch також став повільнішим. Для MXNet результати схожі на результати з рис. 17.

За результатами досліджень було проведено оцінку фреймворків за пам'яттю і швидкістю, де 1 – найгірша оцінка, а 5 – найкраща, 0 – не проводився тест (табл. 1 і 2). Умовні позначення у табл. 1: «A+B» – сума матриць, «A dot B» – множення матриць, «A.T» – транспонування матриць.

Таблиця 1

Оцінка фреймворків за швидкістю

	A+B	A dot B	Reduce mean	Reduce sum	ReLU	Sigm	A.T	Сума
Tensorflow	1	1	3	3	2	2	2	14
MXNet	5	5	5	5	5	5	4	34
PyTorch	2	2	3	2	3	2	5	19
CNTK	0	2	2	3	2	3	0	12

Таблиця 2

Оцінка фреймворків за пам'яттю

Tensorflow	4
MXNet	5
PyTorch	1
CNTK	1

Висновки

За результатами тесту базових операцій було визначено, що найшвидшим за виконанням і найвигіднішим зі пам'яттю виявився фреймворк MXNet. Варто відмітити, що саме символічний підхід працював швидше, оскільки під час такого підходу у пам'яті зберігаються лише ті дані, які необхідні, а не абсолютно усі обчислення. Tensorflow виявився найповільнішим, проте виконання операцій за допомогою цього фреймворку не призводило до витоку пам'яті, операції завжди проводились до самого кінця, хоч і зі збільшенням розмірності задачі це відбувалось повільно. Імперативні підходи у CNTK і PyTorch виявилися дуже невигідними за пам'яттю, хоча на малих розмірностях задач в деяких випадках PyTorch був навіть швидший за імперативний підхід у MXNet.

У подальшому можна порівняти також символічні підходи у PyTorch і CNTK із зібраними результатами. Особливо цікавим буде прослідкувати за затратами по пам'яті, оскільки символічний підхід має сильно його знижувати. Також варто провести тести на GPU та порівняти, який саме з фреймворків справився найкраще на графічному процесорі.

Література

1. TensorFlow [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.tensorflow.org/ (дата звернення 01.05.2018). – Назва з екрана.

2. MXNet: A scalable deep learning framework [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://mxnet.incubator.apache.org/ (дата звернення 01.05.2018). – Назва з екрана.

3. The Microsoft Cognitive Toolkit [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://docs.microsoft.com/en-us/cognitive-toolkit/ (дата звернення 01.05.2018). – Назва з екрана

4. PyTorch [Електронний ресурс] : [Веб-сайт]. – Режим доступу: http://pytorch.org/ (дата звернення 01.05.2018). – Назва з екрана

5. Deep Learning Programming Style [Електронний ресурс] : [Веб-сайт]. – Режим доступу:https://mxnet.incubator.apache.org/architecture/program_model.html (дата звернення 01.05.2018). – Назва з екрана.

6. Deep Learning Part 1: Comparison of Symbolic Deep Learning Frameworks [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.r-bloggers.com/deep-learning-part-1-comparison-of-symbolic-deep-learning-frameworks/ (дата звернення 01.05.2018). – Назва з екрана.

7. A COMPARISON OF DEEP LEARNING FRAMEWORKS [Електронний ресурс] : [Веб-сайт]. – Режим доступу: https://www.exastax.com/deep-learning/a-comparison-of-deep-learning-frameworks/ (дата звернення 01.05.2018). – Назва з екрана.

8. Grad backward memory leak? [Електронний ресурс]: [Веб-сайт]. – Режим доступу: https://github.com/pytorch/pytorch/issues/3824 (дата звернення 01.05.2018). – Назва з екрана.